RDT SITE OFFICE
ANL-Idaho

SOSR
Rec. 11-25-69
[illegible] DM
[illegible]
[illegible]
[illegible]
File [illegible]

ANL-7621

Argonne National Laboratory

GEDANKEN: A Simple Typeless Language Which Permits Functional Data Structures and Coroutines

by

John C. Reynolds

The facilities of Argonne National Laboratory are owned by the United States Government. Under the terms of a contract (W-31-109-Eng-38) between the U. S. Atomic Energy Commission, Argonne Universities Association and The University of Chicago, the University employs the staff and operates the Laboratory in accordance with policies and programs formulated, approved and reviewed by the Association.

MEMBERS OF ARGONNE UNIVERSITIES ASSOCIATION

The University of Arizona	Kansas State University	The Ohio State University
Carnegie-Mellon University	The University of Kansas	Ohio University
Case Western Reserve University	Loyola University	The Pennsylvania State University
The University of Chicago	Marquette University	Purdue University
University of Cincinnati	Michigan State University	Saint Louis University
Illinois Institute of Technology	The University of Michigan	Southern Illinois University
University of Illinois	University of Minnesota	University of Texas
Indiana University	University of Missouri	Washington University
Iowa State University	Northwestern University	Wayne State University
The University of Iowa	University of Notre Dame	The University of Wisconsin

LEGAL NOTICE

This report was prepared as an account of Government sponsored work. Neither the United States, nor the Commission, nor any person acting on behalf of the Commission:

A. Makes any warranty or representation, expressed or implied, with respect to the accuracy, completeness, or usefulness of the information contained in this report, or that the use of any information, apparatus, method, or process disclosed in this report may not infringe privately owned rights; or

B. Assumes any liabilities with respect to the use of, or for damages resulting from the use of any information, apparatus, method, or process disclosed in this report.

As used in the above, "person acting on behalf of the Commission" includes any employee or contractor of the Commission, or employee of such contractor, to the extent that such employee or contractor of the Commission, or employee of such contractor prepares, disseminates, or provides access to, any information pursuant to his employment or contract with the Commission, or his employment with such contractor.

Printed in the United States of America
Available from
Clearinghouse for Federal Scientific and Technical Information
National Bureau of Standards, U. S. Department of Commerce
Springfield, Virginia 22151
Price: Printed Copy $3.00; Microfiche $0.65

ANL-7621
Mathematics and
Computers

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

GEDANKEN: A Simple Typeless Language Which Permits Functional Data Structures and Coroutines

by

John C. Reynolds

Applied Mathematics Division

September 1969

TABLE OF CONTENTS

GEDANKEN - A SIMPLE TYPELESS LANGUAGE WHICH PERMITS FUNCTIONAL DATA STRUCTURES AND COROUTINES

by

John C. Reynolds

ABSTRACT

GEDANKEN is a simple, idealized programming language with the following characteristics: (1) Any value which is permitted in some context of the language is permissible in any other meaningful context. In particular, procedures and labels are permissible results of functions and values of variables. (2) Assignment and indirect addressing are formalized by introducing values, called references, which in turn possess other values. The assignment operation always affects the relation between some reference and its value. (3) All compound data structures are treated as functions. (4) Type declarations are not permitted.

The functional approach to data structures and the use of references insure that any process which accepts some data structure will accept any logically equivalent structure, regardless of its internal representation. More generally, any data structure may be implicit, i.e., it may be specified by giving an arbitrary algorithm for computing or accessing its components. The existence of label variables permits the construction of coroutines, quasi-parallel processes, and other unorthodox control mechanisms.

A variety of programming examples illustrates the generality of the language. Its simplicity is demonstrated by a concise formal definition, in which abstract programs are treated as GEDANKEN data structures, and an interpreter for these structures is given in GEDANKEN itself.

A portion of this material has been submitted to the *Communications of the ACM*.

I. INTRODUCTION

Even a cursory acquaintance with modern programming languages suggests that the simultaneous achievement of simplicity and generality in language design is a serious unsolved research problem. This paper describes a simple and somewhat idealized language, called GEDANKEN, which has developed out of an attempt to attack this problem.

It must be emphasized that GEDANKEN is not intended to be a generally useful programming language, although it could be effective in situations where a fair degree of object program inefficiency is tolerable. Its major purpose is to illustrate two basic principles which the author believes may be valid for the design of more complex and practical languages. This motivation is reflected in its name, which is meant as an analogy to gedanken experiments in physics.

The two principles underlying the design of GEDANKEN are the following:

(1) Completeness. Any value which is permitted in some context of the language is permissible in any other meaningful context. In particular, procedures and labels are permitted to be results of functions or values of references (e.g., variables), without imposing restrictions in order to maintain a stack discipline for run-time storage allocation.

(2) The Reference Concept. To formalize the mechanisms of indirect addressing and assignment, a concept of reference is introduced which is somewhat similar to that used in the Amsterdam proposal for ALGOL 68.(1) Specifically, among the possible values which may occur in a GEDANKEN program are objects called references, which in turn possess other values. The assignment operation always affects the relation between some reference and its value.

We will show that these design principles have the following consequences:

(1) The existence of function-returning and reference-returning functions allows all compound data structures to be treated as functions. Thus, for example, a one-dimensional ALGOL-like array would be treated as a function whose domain was a finite consecutive set of integers and which mapped each element of this domain into a unique reference. This approach insures that any process which accepts some data structure will accept any logically equivalent structure, regardless of its internal representation. More generally, any data structure may be _implicit_, i.e., it may be specified by giving an arbitrary algorithm for computing or accessing its components.

(2) The existence of label variables permits the construction of co-routines, quasi-parallel processes, and other unorthodox control mechanisms. This is a direct consequence of not imposing a stack discipline on the program control information.

Some further design decisions have been made to achieve simplicity and theoretical tractability at the expense of efficiency and practicality. In particular:

(1) Declarations are not allowed to restrict the value range of identifiers, references, or function results. Languages with this property are usually called "typeless," although the types of values may be tested during execution.

(2) GEDANKEN does not provide a variety of common features which enhance the conciseness of a language without expanding the range of programs which can be expressed. Thus for example, infix arithmetic operators, **for** statements, and factored declarations are not provided.

(3) For brevity, floating-point values and operations have been omitted from the version of GEDANKEN described in this paper.

Items (2) and (3) are not conceptually serious deficiencies, since they can be overcome by extending the language in an obvious and well-understood manner. However, item (1) is indicative of a serious theoretical problem: how to develop a facility for type declaration which will permit concise and efficient data representations without destroying the generality of the language.

The design of GEDANKEN has obviously been influenced by a number of earlier languages, including ALGOL 60,(2) LISP,(3) EULER,(4) and ALGOL 68.(1) There is also an unusually close relation between GEDANKEN and the language PAL, described by A. Evans.(5) The similarities of these languages are largely the result of convergent independent evolution, but the existence of sequence expressions and their use in treating all functions as functions of a single variable are direct borrowings from PAL. (However, PAL does not treat sequences as special cases of functions, nor does it utilize the reference concept.)

In the sequel, we give both an informal description of GEDANKEN and a formal definition. The generality of the language will be demonstrated by a variety of examples; its simplicity is evident from the conciseness of the formal definition.

II. AN INFORMAL DESCRIPTION OF GEDANKEN

The Universal Value Set

The principle of completeness and the absence of type declarations in GEDANKEN implies that the values of all identifiers and references, as well as the results of all functions, may range freely over a single universal set. This set contains the following types of data:

<u>Primitive Data</u>: Integers, Booleans, Characters, Atoms

<u>Nonprimitive Data</u>: Functions, References, Label values

A complete set of basic predicate functions is available for testing the type of a datum.

Among the primitive data only atoms are unusual. They are similar to atoms in LISP, except that they lack property lists and print names. There is a basic function ATOM which produces a new distinct atom each time it is executed, and atoms may be tested for equality. Two particular atoms, denoted by the pre-defined identifiers LL and UL, play a special role in the language.

All functions accept a single argument and produce a single result, with possible side effects. Proper procedures are treated as functions which execute side effects but produce an irrelevant result. A number of basic functions are provided which may be used without being defined (see Appendix); additional user-defined functions are created by the evaluation of various expressions.

The functional approach to data structures gives special importance to functions called vectors, whose domain is a finite set of consecutive integers. Given a vector, it is useful to be able to determine the limits of its domain. In several languages, these limits are obtained by applying certain basic functions to the vectors, but this approach destroys the pure functionality of the vectors. Thus in GEDANKEN, the limits are obtained by applying the vector to two special arguments, the atoms LL and UL.

More precisely, a function F is called a vector if: (1) Its domain (the set of arguments for which the function terminates without an error stop) includes the atoms LL and UL, (2) The values of F(LL) and F(UL) are integers such that F(UL) ≥ F(LL) - 1. (3) The domain includes all integers I such that F(LL) ≤ I ≤ F(UL).

If F is a vector, then the integer F(UL) - F(LL) + 1 is called the length of F, and, for each F(LL) ≤ I ≤ F(UL), the value of F(I) is called the Ith component of F. If F is a vector and F(LL) = 1 then F is called a sequence.

It should be emphasized that any function satisfying the above definition is a legitimate vector, regardless of the method by which it computes its components or limits. Indeed, there is no basic predicate function which tests whether an arbitrary function is a vector, since such a predicate would be noncomputable. However, the language provides certain operations whose result will always be a vector which does not cause side effects.

Sequences are used to reduce functions of several arguments to functions of a single argument. Thus a function of k arguments is treated as a function of a single argument that is a sequence of length k.

References and _label_ values will be discussed later.

Syntax

A GEDANKEN program is a sequence of _tokens_ separated by zero or more blanks, with at least one blank used as a separator whenever the juxtaposition of two tokens would otherwise be ambiguous. The tokens themselves are sequences of characters classified as follows:

Constants: digit strings (denoting integers), quoted strings

Reserved Words: AND, OR, IF, THEN, ELSE, CASE, OF, IS, ISR

Identifiers: All other alphanumeric strings beginning with a letter

Punctuation Tokens: λ , = : ( ) ; :=

Certain identifiers, called _predefined_, have standard meanings. These include TRUE, FALSE, LL, UL, and QUOTECHAR, which denote fixed items of primitive data; ERROR, which denotes a basic label value causing program termination; and the names of all basic functions. The meanings of predefined identifiers, but not reserved words, may be overridden by declarations.

The syntax of GEDANKEN is specified by a context-free grammar over an infinite vocabulary of tokens rather than a finite vocabulary of characters. We state this grammar in an extension of BNF in which the notation $\{\alpha\}^*$ is used to indicate an arbitrary number (including zero) of occurrences of the string α:

<exp$_0$> ::= <constant> | <identifier> | (<block>)

<exp$_1$> ::= <exp$_0$> | <function designator>

<function designator> ::= <exp$_0$><exp$_1$>

<exp$_2$> ::= <exp$_1$> | <exp$_1$> = <exp$_2$>

<exp$_3$> ::= <exp$_2$> | <exp$_2$> AND <exp$_3$>

<exp$_4$> ::= <exp$_3$> | <exp$_3$> OR <exp$_4$>

<exp$_5$> ::= <exp$_4$> | <conditional exp> | <lambda exp> | <exp$_4$> := <exp$_5$>

<conditional exp> ::= IF <exp$_6$> THEN <exp$_6$> ELSE <exp$_5$>

<lambda exp> ::= λ <pform$_0$><exp$_5$>

<exp$_6$> ::= <exp$_5$> | <sequence exp> | <case exp>

<sequence exp> ::= <empty> | <exp$_5$>, <exp$_5$> {, <exp$_5$>}*

<case exp> ::= CASE <exp$_6$> OF <exp$_5$> {, <exp$_5$>}*

<pform$_0$> ::= <identifier> | (<pform$_1$>)

<pform$_1$> ::= <pform$_0$> | <sequence pform>

<sequence pform> ::= <empty> | <pform$_0$>, <pform$_0$> {, <pform$_0$>}*

<decl> ::= <pform$_1$> IS <exp$_6$>

<recursive decl> ::= <identifier> ISR <lambda exp>

<label> ::= <identifier> :

<statement> ::= {<label>}* <exp$_6$>

<block> ::= {<decl>;}* {<recursive decl>;}* {<statement>;}* <statement>

<program> ::= <block>

Phrases of the classes exp$_0$, ... , exp$_6$ are all called expressions; the subscripts serve only to distinguish levels of precedence. Similarly both pform$_0$ and pform$_1$ are called parameter forms. It should be noted that a block can consist of a single expression, so that any expression can be parenthesized (without changing its semantics).

Applicative Semantics of GEDANKEN

Following Evans,(5) we divide the semantics of GEDANKEN into an applicative part, involving the evaluation of expressions and the creation and application of functions, and an imperative part, involving references, assignments, labels, and jumps. We first consider the applicative part, which is a complete and nontrivial language in itself.

The application of a function to an argument is performed by evaluating a function designator:

$$\text{<function designator> ::= } \underbrace{\text{<exp}_0\text{>}}_{\text{function part}} \quad \underbrace{\text{<exp}_1\text{>}}_{\text{argument part}}$$

Such an expression is evaluated by first evaluating its function part and its argument part to obtain values v_f (which must be a function) and v_a, and then applying the function v_f to the argument v_a, taking the result of v_f to be the value of the function designator.

The syntax allows the usual composition of functions to be written without parentheses, e.g., SIN(SQRT(X)) may be written as SIN SQRT X. On the other hand, parentheses are needed to apply a function which is the result of another function, e.g., (X 3) 4 causes the function X to be applied to 3 and then the result of X to be applied to 4.

Functions are created by evaluating lambda expressions:

$$\text{<lambda exp> ::= } \lambda \text{ <pform}_0\text{> } \underbrace{\text{<exp}_5\text{>}}_{\text{body}}$$

For the moment we limit ourselves to the case where the parameter form is a single identifier. Then the value of a lambda expression is the function whose result is obtained by evaluating the body after binding the (identifier which is the) parameter form to the argument of the function. To complete this definition we must specify the binding of any free identifiers which occur in the body but not in the parameter form. Such identifiers are bound to the values which they possess when the lambda expression is evaluated (which may not be the same as their values later when the resulting function is applied). This type of binding is called FUNARG binding in LISP and is similar to the binding of free identifiers in ALGOL.

Functions which are sequences may also be created by evaluating <u>sequence expressions</u>:

$$\langle\text{sequence exp}\rangle ::= \langle\text{empty}\rangle \mid \langle\text{exp}_5\rangle\ ,\ \langle\text{exp}_5\rangle\ \{,\ \langle\text{exp}_5\rangle\}^*$$

Let n be the number of subexpressions. Then the sequence expression is evaluated by first evaluating its subexpressions to obtain values $v_1, \ldots, v_n$ and then creating a sequence of length n whose ith component (for $1 \leq i \leq n$) is v_i.

Because of their low precedence, sequence expressions are often parenthesized, but the parentheses themselves do not indicate the construction of a sequence. Thus the expressions () and (X, Y) both create sequences, but (X) has the same value as X. There is no sequence expression which produces a sequence of length one, but such sequences can be produced by the basic function UNITSEQ, which returns a sequence whose only component is the value of its argument.

As noted earlier, a function of n variables ($n \neq 1$) is treated in GEDANKEN as a function of a sequence of length n. However, the syntax is arranged to preserve conventional notation. Thus, for example, ADD(X, Y) has its usual effect, but this effect is achieved by creating a sequence out of the values of X and Y and then giving this sequence to ADD as its single argument.

This situation suggests that when a function created by a lambda expression expects to receive a sequence as its argument, the parameter form within ~~the~~ the lambda expression should be able to bind several different identifiers to the components of the sequence. Such a capability is provided by sequence parameter forms:

$$\langle\text{sequence pform}\rangle ::= \langle\text{empty}\rangle \mid \langle\text{pform}_0\rangle, \langle\text{pform}_0\rangle \{, \langle\text{pform}_0\rangle\}^*$$

In general, if a is any value and p is any parameter form, then the binding of p to a is defined recursively as follows:

(1) If p is an identifier, then p is bound to a.

(2) If p has the form (p'), then p' is bound to a.

(3) If p is a sequence parameter form, $p_1, \ldots, p_n (n \neq 1)$, then a, which must be a function, is applied to each integer from 1 to n, and each p_i is bound to the value of a(i).

The combined syntax of sequence expressions and sequence parameter forms is designed to preserved conventional notation for functions of several arguments. Thus in the evaluation of (λ(X, Y) body)(3, 4), X is bound to 3 and Y is bound to 4. However, the sequence argument approach also provides useful unconventional capabilities, e.g., (λ(X, Y) body)(IF P THEN (3, 4) ELSE (5, 6)). More importantly, the ability to bind a single identifier to an entire sequence provides the equivalent of a function with an indefinite number of arguments, e.g., (λX body)(IF P THEN (3, 4) ELSE (5, 6, 7)).

GEDANKEN is similar to EULER(4) in treating all types of unlabelled statements as expressions (some of which are evaluated for their side effects rather than their values). In particular, a (parenthesized) block is a type of expression with a meaningful value:

$$\langle\text{block}\rangle ::= \{\langle\text{decl}\rangle;\}^* \{\langle\text{recursive decl}\rangle;\}^* \{\langle\text{statement}\rangle;\}^* \langle\text{statement}\rangle$$

It is evaluated by first carrying out the bindings indicated by its declarations, recursive declarations, and labels, and then evaluating the statements in order from left to right. If no jumps out of the block occur, the value of the block is the value of the rightmost statement (the values of preceding statements are ignored).

A <u>declaration</u>:

```
<decl> ::= <pform_1> IS <exp_6>
```

is executed by evaluating the expression on its right and then binding the parameter form to the value of this expression. A sequence of declarations is executed from left to right, so that the expression in each declaration "feels" only the bindings caused by preceding declarations on the left.

Unfortunately, since a declaration does not bind its own right side, it cannot be used easily to define a recursive function. Thus for example,

```
FACT IS λN IF N = 0 THEN 1 ELSE MULTIPLY(N, FACT SUBTRACT(N,1))
```

does not define a recursive function since the occurrence of FACT on the right side is not bound to the value of the lambda expression containing it. To overcome this problem, <u>recursive declarations</u> are provided:

```
<recursive decl> ::= <identifier> ISR <lambda exp>
```

A recursive declaration binds the identifier on its left to the value of the lambda expression on its right, but in computing this value, the free identifiers in the lambda expression are not bound to their values at the time the lambda expression is evaluated. Instead, these free identifiers are bound to their values after all bindings (including label bindings) in the block containing the recursive declaration have been completed. This allows the use of several recursive declarations to define a family of recursive functions which call each other. It also allows a recursive function to jump to some statement in the block in which it is defined.

Conditional expressions have the same meaning as in ALGOL. A <u>case expression</u> of the form CASE e_0 OF e_1, ..., e_n is executed by first evaluating e_0 to obtain a value i; if i is an integer satisfying $1 \le i \le n$, the result of the case expression is obtained by evaluating e_i; if i is LL or UL the result is 1 or n; all other values of i give an error. The remaining forms of expressions can all be regarded as abbreviations (except for coercion):

$$e_1 = e_2 \Rightarrow \text{EQUAL}(e_1, e_2)$$
$$e_1 \text{ AND } e_2 \Rightarrow (\text{IF } e_1 \text{ THEN } e_2 \text{ ELSE FALSE})$$
$$e_1 \text{ OR } e_2 \Rightarrow (\text{IF } e_1 \text{ THEN TRUE ELSE } e_2)$$
$$e_1 := e_2 \Rightarrow \text{SET}(e_1, e_2)$$

where EQUAL denotes a basic equality function and SET denotes a basic assignment function (to be defined later). EQUAL tests the equality of primitive data, but if either component of its argument is a function or a label, it will return FALSE. Its action on references will be described later.

The meaning of constant tokens which are quoted strings requires some explanation. A single quoted character denotes the corresponding primitive datum, but a string of any other length denotes a sequence whose components are the characters of the string (with quotation marks deleted).

<u>Data Structures in Applicative GEDANKEN</u>

Even the applicative part of GEDANKEN is sufficient to demonstrate the power and flexibility which are obtained by treating compound data structures as functions.

As a first example, consider LISP-like list structures. We wish to define functions in GEDANKEN which are equivalent to the LISP functions CONS, CAR, and CDR. The two-field list cell produced by CONS can be considered to be a function whose domain contains two elements (e.g., 1 and 2) and which maps these elements into the values of its CAR and CDR fields. This viewpoint leads directly to the definitions:

```
        CONS IS λ(X, Y) λZ IF Z = 1 THEN X ELSE Y;
        CAR IS λX X 1;
        CDR IS λX X 2;
```

These definitions imply an ability to do list processing without the use of special basic functions. In a conventional language (e.g., compiled LISP 1.5[3] or various extensions of ALGOL[1,6]) user-defined functions are restricted so that storage for the values of their identifiers obeys a stack discipline. In this situation list structures, which do not obey a stack discipline, must be allocated in a separate storage area, and basic functions or operations must be provided for accessing this area. But in GEDANKEN, the user may develop list-processing by defining function-returning functions (such as CONS above) which violate a stack discipline. In effect, all storage is potentially list-structured.

Although the above approach is workable and theoretically attractive, it is more convenient to use sequence expressions to create list elements, i.e., to write (X, Y) instead of CONS(X, Y), X 1 instead of CAR X, and X 2 instead of CDR X. Following this approach, we introduce lists into GEDANKEN by first creating an atom to denote the empty list:

```
        NIL IS ATOM();
```

and then defining a list to be either the atom NIL or a sequence of length two whose second component is a list. Then the following functions will produce the length of a list, find the ith element of a list, and append one list to another:

```
LISTLENGTH ISR λL IF L = NIL THEN 0 ELSE INC LISTLENGTH L 2;
LISTELEM ISR λ(I, L) IF L = NIL THEN GOTO ERROR
    ELSE IF I = 1 THEN L 1 ELSE LISTELEM(DEC I, L 2);
APPEND ISR λ(X, Y) IF X = NIL THEN Y ELSE (X 1, APPEND(X 2, Y));
```

Here INC and DEC are basic functions which increase or decrease an integer by one.

As a second example, we consider one-dimensional arrays. We have already defined a type of function called a vector which is the analogue of a one-dimensional array, and we have introduced sequence expressions for creating vectors. But a sequence expression can only produce a vector which is a sequence, and it is inconvenient for producing very long vectors. What is needed is a function which will produce a vector from a functional specification of its components.

Thus we wish to define a function, called VECTOR, which will accept another function, tabulate its values over a finite range, and return a lookup function for the resulting table. More precisely, VECTOR accepts an argument (L, U, F), where L and U are integers and F is a function. If U < L, VECTOR produces an empty vector V such that V(LL) = L and V(UL) = L - 1. Otherwise, VECTOR evaluates F(I) for each integer I between L and U inclusive, and produces a vector V such that V(LL) = L, V(UL) = U and for L ≤ I ≤ U, V(I) is the value of F(I). The following definition meets this specification:

```
VECTOR ISR λ(L, U, F)
IF GREATER(L, U) THEN
    λ I IF I = LL THEN L ELSE IF I = UL THEN DEC L ELSE GOTO ERROR
ELSE (V IS VECTOR(L, DEC U, F); T IS F U;
    λ I IF I = UL THEN U ELSE IF I = U THEN T ELSE V I);
```

It is evident that this function, although theoretically correct, will be extremely inefficient in any reasonable implementation. For this reason, a basic function VECTOR is provided which is defined to be equivalent to the function above (except for coercion).

To clarify this question of efficiency it is necessary to consider possible implementations of GEDANKEN, without going into unnecessary detail. In a simple implementation, functions will possess two distinct internal representations: If a function is created by evaluating a lambda expression, it will be represented by a "lambda record" containing a pointer to code which was compiled from the lambda expression, plus values for each free identifier in the lambda expression. On the other hand, if a function is created by evaluating a sequence expression or by the application of VECTOR, it will be represented by a "vector record" containing range limit and indexing information, plus a contiguous array of component values (or perhaps a pointer to such an array). It is evident that the above definition of VECTOR (as opposed to the basic function) would yield a vector whose internal representation was a linked list of lambda records (each containing one component value) rather than a contiguous array.

We may now illustrate our assertion that any process which accepts some data structure will accept any logically equivalent structure. Suppose that P is a function which expects a sequence as its argument, and that we wish to give it a sequence whose ith component is the ith element of a list L. This can be done in a conventional manner by evaluating P VECTOR(1, LISTLENGTH L, λ I LISTELEM(I, L)), which copies the elements of L into a contiguous array. But it is also possible to evaluate P MAKESEQFROMLIST L, where

```
    MAKESEQFROMLIST IS λ L
    λ I IF I = LL THEN 1 ELSE IF I = UL THEN LISTLENGTH L ELSE LISTELEM(I, L);
```

MAKESEQFROMLIST does not copy the components of L; instead, it produces an implicit sequence which will look up the appropriate element of L each time one of its components is accessed.

It is equally possible to produce an implicit list from a sequence:

```
    MAKELISTFROMSEQ ISR λ S MLFS1(1, S);
    MLFS1 ISR λ(I, S) IF GREATER(I, S UL) THEN NIL
        ELSE λ K (CASE K OF S I, MLFS1(INC I, S));
```

(Here MLFS1 is a subsidiary function which produces an implicit list from the subsequence of S that begins with the Ith component.)

So far, the data structures we have shown all have the limitation that once a structure has been created, its components or elements cannot be altered. To overcome this limitation we must introduce the imperative aspects of GEDANKEN.

References

It is well known that the introduction of assignment into an applicative language requires a careful distinction between values and objects which possess values (variously called addresses, pointers, references, or names). In GEDANKEN we have chosen to formalize this distinction by introducing a concept of a reference which is rather similar to that used in ALGOL 68.(1) It is not clear that this is the best approach, but it is flexible and conceptually simple, and it combines cleanly with the principle of completeness.

In applicative GEDANKEN the only entities which possess values are identifiers; an identifier is said to be bound to its value. It would be possible to define assignment as an operation which changes the binding of an identifier, but this would destroy the "nesting" property of binding. Moreover, if assignment is considered as a function (with side effects), then the principle of completeness would be violated, since an identifier would be a possible argument for the assignment function but would not be a permissible value in any other context.

Thus we introduce a second type of entity, called a <u>reference</u>, which possesses a value. But unlike an identifier, a reference can also be a value, possessed by either an identifier or a reference. The assignment operation is then defined to alter the relation between a reference and its value rather than an identifier and its value.

Basically, references are manipulated by three basic functions: REF, SET, and VAL. REF X returns a new distinct reference which is initialized to possess the value X. SET(R, X) (which can be abbreviated R := X) causes R (which must be a reference) to possess the value X and then returns X. VAL R returns the value possessed by R (which must be a reference).

The following illustrates the reference concept: Under the scope of the declaration X IS 3, the identifier X is bound to the integer 3, and this binding cannot be altered by assignment. Indeed, evaluation of the expression X := 4 would give an error, since 3 is not a reference. Analogously, under the scope of the declaration X IS REF 3, the identifier X is bound to the reference created by REF, and this binding cannot be changed by assignment. But now evaluation of X := 4 is legitimate, and causes the value possessed by the reference bound to X to change from 3 to 4. Thus, in the execution of the block

```
(X IS REF 3; VAL X = 3; X := 4; VAL X = 4)
```

both equality predicates will be true.

The major difficulty with references is the frequent necessity for using the function VAL. Thus under the scope of the declarations X IS REF 3;

Y IS REF 4; one must write ADD(VAL X, VAL Y) rather than ADD(X, Y), since ADD acts upon integers rather than references. To alleviate this difficulty, we adopt a coercion convention, in which references are automatically replaced by their values in contexts in which they would otherwise be meaningless.

Specifically, we define the function

```
COERCE ISR λ X IF ISREF X THEN COERCE VAL X ELSE X;
```

(which is available as a basic function), and define the phrase "to coerce X" to mean the replacement of X by COERCE X. Then the following conventions are established:

(1) All basic functions which would otherwise be meaningless coerce their argument or the appropriate components of their arguments. For example, ADD(X, Y) is equivalent to ADD(COERCE X, COERCE Y), but ISREF X is not equivalent to ISREF COERCE X, nor VAL X to VAL COERCE X.

(2) REF X coerces X, SET(R, X) (and therefore R := X) coerces X, and EQUAL(X, Y) (and therefore X = Y) coerces both X and Y. Since these functions would each be meaningful for references without coercion, analogous noncoercing functions, named NCREF, NCSET, and NCEQUAL, are also provided. NCREF and NCSET permit references to possess values which are also references. (Note that COERCE NCSET(R, R) never terminates.) NCEQUAL can be used to determine whether two values are the same reference.

(3) Conditional and case expressions coerce the values of their leftmost subexpressions.

(4) Expressions involving AND and OR coerce the values of both of their subexpressions.

(5) A function designator coerces the value of its function part.

(6) When a sequence parameter form $p_1, \ldots, p_n$ is bound to a value a, each p_i will be bound to (COERCE a)(i).

(7) Vectors which are created by evaluating sequence expressions or by application of the basic functions VECTOR or UNITSEQ will coerce their argument.

Data Structures with Imbedded References

The utility of the reference concept becomes apparent when reference-returning functions are used to imbed references within data structures, yielding structures which can be altered by assignment.

This approach provides precise control over the ways in which data structures can be altered. Thus the GEDANKEN equivalent of an ALGOL-like one-dimensional array is a vector whose components are references, e.g.,

```
X IS VECTOR(1, 100, λ I REF 0);
```

Under the scope of this declaration, assignment can be made to the components of X, e.g., X(7) := 10, but not to X itself. In particular, the subscript limits X LL and X UL are fixed by the declaration.

On the other hand, the equivalent of a string variable is provided by a reference whose value is a vector:

```
S IS REF VECTOR(1, 100, F);
```

Here assignment can be made to S itself (possibly changing the subscript limits) but not to the components.

A second consequence of the reference concept is the ability to define data structures or sets of data structures which share elements, in the sense that assignment to one element will affect another. Suppose we wish to define a square matrix M. We could define M as a vector of vectors, i.e.,

```
M IS VECTOR(1, 10, λ I VECTOR(1, 10, λ J REF 0));
```

but this leads to the inconvenience of referring to an element of M by (M I) J.

It is more natural to define M as a reference-returning function of pairs of integers:

```
        M IS(M1 IS VECTOR(1, 10, λ I VECTOR(1, 10, λ J REF 0));
           λ(I, J) (M1 I) J);
```

so that an element is referred to as M(I, J). Now consider the additional declarations:

```
    MT IS  λ(I, J) M(J, I); MD IS λ I M(I, I);
```

Here MT and MD denote the transpose and diagonal of M, in the sense that assignment to an element of one matrix affects the corresponding elements of the others.

Elements may also be shared within the same data structure. For example,

```
        S IS (S1 IS VECTOR(1, 10, λ I VECTOR(1, I, λ J REF 0));
           λ(I, J) IF NOT GREATER(J, I) THEN (S1 I) J ELSE (S1 J) I);
```

defines a symmetric matrix in which assignment to S(I, J) also alters S(J, I).

The imbedding of references in list structures also provides control over the ways in which these structures may be altered. This is in contrast with a language such as LISP, where one must choose between using a purely applicative language (pure LISP) in which list structures can never be altered, and a language (full LISP 1.5) in which every field of every list cell can be altered.

As an example, consider a property list, which is a list of property-value pairs subject to two operations: The value paired with a given property may be looked up, or the value paired with a given property may be changed, adding a new pair to the list if the property is not already present. It is evident that references must occur in the property list at two points: Each value must be a reference, so that it can be changed, and the entire list must be a reference, so that new pairs can be added.

The following function manipulates such property lists. Given a property P and a (reference to a) property list L, PROPVAL(P, L) search L for an occurrence of P. If P is found, the reference paired with P is returned. Otherwise, a pair consisting of P and a new reference (initialized to zero) is added to L, and the new reference is returned. The argument P is coerced.

```
        PROPVAL IS λ(P, L)
            (P IS COERCE P;
            SEARCHL ISR λ X
               IF X = NIL THEN
                  (NEWV IS REF 0; L := ((P, NEWV), VAL L); NEWV)
               ELSE IF (X 1) 1 = P THEN (X 1) 2 ELSE SEARCHL X 2;
            SEARCHL VAL L);
```

A call of this function can occur on either side of an assignment operation; on the right side it will act to look up a value, on the left side it will act to alter a value.

A further step can be taken by viewing the property list itself as a reference-returning function which accepts a property and produces a reference to the corresponding value. The following function (of no arguments) creates such functional property lists:

```
        MAKEPROPLIST IS λ() (L IS REF NIL; λ P PROPVAL(P, L));
```

Essentially, each execution of MAKEPROPLIST creates a new instance of PROPVAL, with L bound to a private "own variable." Since a property can be any primitive datum, a functional property list is similar to a reference-valued vector, except that it has an indefinite domain. Indeed, functional property lists can be used to provide an efficient implementation of sparse vectors.

As a final example of the use of references, suppose that READ is a function such that each execution of READ produces the next item of data from some input stream, and that we wish to produce an implicit list of the successive items in the stream. The following function (of no arguments) produces such a list:

```
          MAKERLIST ISR λ()
              (B IS REF 0; λ I
                 (IF B = 0 THEN B := (READ(), MAKERLIST()) ELSE (); B I));
```

The result of MAKERLIST is an implicit list (whose implicit length is infinite) which only calls READ as items of data are actually needed, and only stores previously read items which are still accessible.

Implicit References

The utility of implicit data structures suggests the introduction of an analogous facility for references. Thus we extend the concept of a reference to include an implicit reference, which may carry out an arbitrary computation each time it is set or evaluated.

An implicit reference is created by executing the basic function IMPREF(SETF, VALF), where SETF and VALF must be functions of one and zero arguments respectively. Each execution of IMPREF creates a distinct implicit reference, and these implicit references satisfy the predicate ISREF and are coerced in the same manner as conventional references. But the effect of SET or VAL on an implicit reference is to execute SETF or VALF. Specifically, if R is the result of IMPREF(SETF, VALF) then

```
        NCSET(R, X) = (SETF X; X)
        SET(R, X) = (X IS COERCE X; SETF X; X)
        VAL R = VALF ()
```

To illustrate the use of implicit references, we consider the problem of protecting a reference-valued vector. Suppose that P is a function which accepts a vector whose components are references. We wish to apply P to such a vector V, but to protect the components of V from being affected by P, i.e., we want these components to revert to their original values after the execution of P is finished. The simplest approach is to copy V by executing P VECTOR(V LL, V UL, λ I REF V I), but this will be inefficient if V is large and only a few components are reset by P. An alternative approach is to maintain a "change list" of the components of V which have been altered by P. This may be done by executing P PSEUDOCOPY V, where

```
PSEUDOCOPY IS λ V
      (CL IS REF NIL;
      SEARCHCL ISR λ(X, I, F, G) IF X = NIL THEN G()
         ELSE IF (X 1) 1 = I THEN F (X 1) 2 ELSE SEARCHCL(X 2, I, F, G);
      λ I (I IS COERCE I;
         IF I = LL THEN V LL ELSE IF I = UL THEN V UL
         ELSE IF NOT ISINTEGER I OR GREATER(V LL, I) OR GREATER(I, V UL)
            THEN GOTO ERROR
         ELSE IMPREF(
            λ X  SEARCHCL(VAL CL, I, λ R NCSET(R, X),
               λ() CL := ((I, NCREF X), VAL CL)),
            λ() SEARCHCL(VAL CL, I, VAL, λ() VAL V I))));
```

CL is a reference to the change list, which is a list of pairs, each containing an integer argument of some altered component and a reference to the current value of that component. SEARCHCL is a subsidiary function which searches a change list X for a pair beginning with the integer I. If such a pair is found, SEARCHCL returns the value of F applied to the reference paired with I; otherwise SEARCHCL returns the value of G, which is a function of no arguments. The noncoercing functions NCSET and NCREF are used to allow P to set a component of V to a reference.

Label Values

The final type of data used in GEDANKEN is the label value. These values are created during execution of a block containing labelled statements, and are used as arguments to the basic function GOTO, which never returns but instead causes a transfer of control to the computational state represented by the label value.

During the execution of a block, immediately before the first statement of the block is executed, each identifier that labels a statement in the block is bound to a label value, which contains two items of information: the sequence of statements beginning with the labelled statement, and the current status of the computation. The informal notion of current status will be made more precise in the next chapter. For the present, we note that it contains the identifier bindings appropriate for executing statements in the block (including label bindings), plus all information necessary to complete the computation after the block has exited. However, it does not include the mapping of references into their values.

Execution of the basic function GOTO, whose argument must be a label value, causes the current status of the computation to be replaced by the status stored in the label value, and then causes execution to continue with the statement sequence stored in the label value.

As in ALGOL, this approach permits jumps within the same block or to higher-level blocks. But the fact that label values can be possessed by references or returned by functions provides additional capabilities, including the ability to jump back into a block after it has been exited from. It is this capability which allows the construction of coroutines.

Coroutines

A coroutine is a procedure which can relinquish control to its calling program and later be reactivated to continue computation. The simplest situation is that of two procedures, each of which treats the other as a subroutine.

As an example, suppose that COMPILE is a procedure which produces a succession of data items called instructions, outputting each instruction by applying a function OUT, and that ASSEMBLE is a procedure which accepts a succession of instructions, inputting each instruction by applying a function IN. If OUT and IN are arguments to COMPILE and ASSEMBLE respectively, we have

```
        COMPILE ISR λ OUT ( ... OUT X ... );
        ASSEMBLE ISR λ IN ( ... X := IN() ... );
```

We now want to couple these procedures so that ASSEMBLE receives the output of COMPILE. Specifically, we want to run ASSEMBLE until it requests input, then run COMPILE until it produces the required output, then run ASSEMBLE again, etc. The necessary program can be written by using label-valued references which are global to both IN and OUT:

```
        (LC IS REF 0; LA IS REF 0; INST IS REF 0;
        LC := LC1; ASSEMBLE(λ() (LA := LA1; GOTO LC; LA1: VAL INST)); GOTO DONE;
        LC1:  COMPILE (λ X (LC := LC2; INST := X; GOTO LA; LC2:)); GOTO ERROR;
        DONE:);
```

Here LA and LC are label-valued references saving the current states of ASSEMBLE and COMPILE, and INST is a third reference used to hold the instruction being transmitted from COMPILE to ASSEMBLE. If COMPILE finishes while ASSEMBLE is still waiting for another instruction, an error stop occurs.

Nondeterministic Algorithms

Label values in GEDANKEN are closely related to "processes" in simulation languages such as SIMULA;[7] both are mechanisms which allow the state of a suspended computation to be saved as an item of data. The essential difference is that further execution of a computation which was saved as a process causes the process to be updated, while further execution of a computation saved as a label value leaves the label value unchanged. Thus label values can be used to repeatedly initiate execution from the same state.

This capability can be used to program a mode of execution for nondeterministic algorithms[8] in which alternative paths are pursued concurrently. A simple example is nondeterministic parsing. It is fairly straightforward to convert a context-free grammar into a recursive parsing function. Unfortunately, for many grammars this function will contain nondeterministic branches, i.e., points at which a conditional branch must be performed although the current state of the parse is insufficient to determine this branch.

When nondeterministic branches occur, parsing can be accomplished by simulating a finite set of independent parsers, all accepting the same input string and obeying the same program, but with different control states. When a parser encounters a nondeterministic branch, it expands into two separate parsers; when a parser reads an input character which is inconsistent with its control state, it is deleted.

Specifically, we assume that PARSE(IN, AMB, FAIL) is a function which accepts two functions IN and AMB, and a label value FAIL, and returns some representation of a successful parse. The function IN, of no arguments, is called by PARSE to read each character of the input string. The function

AMB, whose argument is a label value, is called to execute a nondeterministic branch; one side of the branch returns from AMB while the other jumps to the label-valued argument. PARSE jumps to the label value FAIL when it encounters an inconsistent character. We assume that PARSE does not set any references, or at least that it does not expect the value of any reference to be preserved across a call of IN or AMB.

The following program carries out the concurrent execution of PARSE, synchronizing the independent parsers by their reading of characters:

```
(C IS REF NIL; W IS REF NIL; R IS REF NIL; CHAR IS REF NIL;
    C := (PARSE( λ() (W := (L1, VAL W); GOTO CONT; L1: VAL CHAR),
                λ L2 (R := (L2, VAL R)), CONT),
        VAL C);
CONT: IF R = NIL AND W = NIL THEN GOTO DONE
        ELSE IF R = NIL THEN (CHAR := READCHAR(); R := W; W := NIL)
        ELSE ();
    (L IS R 1; R := R 2; GOTO L);
DONE: VAL C)
```

Each independent parser is represented by a label value if it has not completed its parse, or by its result if it has completed its parse. The finite set of parsers is maintained by the values of the references C, W, and R. C gives a list of the results of completed parses, W gives a list of label values representing the parsers which are waiting for the next character, and R gives a similar list for the parsers which are ready for execution before reading the next character. The reference CHAR keeps track of the current character, and is updated by the basic function READCHAR. The label CONT is reached whenever execution is to be switched from one parser to another. The final value of the block is the list of completed parses; the input string is ill-formed, well-formed, or ambiguous depending upon whether this list has zero, one, or more than one element.

This approach to parsing is basically the same as that used in the COGENT programming system.(9) It is presented here as an illustration of the generality of GEDANKEN, but it does not represent a significant advance in the field of parsing techniques. Although it is reasonably efficient for a large class of unambiguous grammars (at least if the function PARSE is carefully constructed), certain ambiguous grammars will cause an exponential growth in the number of parsers, and are better treated by other methods, such as that of Earley.(10)

III. A FORMAL DEFINITION OF GEDANKEN

Our approach to the formal definition of GEDANKEN is based on the work of the IBM Vienna group,(11) which is an extension of work by Landin(12) and McCarthy.(13) We will define an abstract syntax for GEDANKEN, show how to translate concrete GEDANKEN programs into abstract programs which satisfy this syntax, and then define the semantics of the language by giving an interpreter which accepts abstract programs. However, we will deviate from the Vienna approach in two particulars:

(1) A definite order of evaluation will be specified for all phrases of abstract programs.

(2) Rather than introducing a special notation for abstract programs or the functions that manipulate them, we will treat abstract programs as GEDANKEN data structures and use GEDANKEN itself to define the functions. To minimize the dangers of circularity inherent in this approach, we will avoid using features of GEDANKEN which are novel or potentially ambiguous, such as label-valued references or the imbedding of references in data structures. Overall, simplicity and clarity will be emphasized at the expense of efficiency.

Abstract Syntax Definition

An abstract program will be a GEDANKEN data structure which is a combination of sequences and records. In conventional languages, a record is a finite collection of fields, each of which is identified by a field name. In GEDANKEN we will use atoms as field names and consider a record to be a function whose domain is a finite set of such atoms and which maps each atom into the corresponding field value.

Following Wirth and Hoare,(6) we assume that the set of all records is partitioned into a finite number of disjoint subsets called classes, and that all records in the same class have the same set of fields. To insure this we require that: (1) The domain of every record contains the atom TYPE, (2) Each record maps TYPE into an atom called its class name, (3) If R1 and R2 are records such that R1 TYPE = R2 TYPE, then R1 and R2 have the same domain.

Moreover we assume that, for a particular field of records in a particular class, the range of possible field values may be restricted. To describe such restrictions we use an expression called a class definition, with the form:

```
        <class definition> ::=
            (CLASS, <class name> {, (<field name>, <range descriptor>)}*)
```

where

```
        <range descriptor> ::= <set name> | SEQ, <set name>
```

The class definition (CLASS, c, (f_1, r_1), ... , (f_n, r_n)) implies that for every record R in class c (i.e., such that R TYPE = c): (1) The domain of R is the set of atoms {TYPE, f_1, ... , f_n}; (2) If r_i is a set name s, then the value of R f_i is a member of the set denoted by s; (3) If r_i has the form SEQ, s, then the value of R f_i is a sequence whose components are all members of the set denoted by s.

A set name may be any of the following: (1) INTCLASS, BOOLCLASS, and CHARCLASS, denoting the sets of integers, boolean values, and characters respectively; (2) UNIVERSAL, denoting the universal value set; (3) A class name, denoting a class of records; (4) A _union name_, denoting the union of sets denoted by other set names. The meaning of _union names_ is described by expressions called _union definitions_, with the form:

```
        <union definition> ::= (UNION, <set name> {, <set name>}*)
```

The union definition (UNION, s_0, s_1, ... , s_n) implies that the set name s_0 denotes the union of the sets denoted by s_1, ... , s_n. Circular union definitions, e.g., (UNION, X, X) are not permitted.

A collection of interrelated class and union definitions, defining various record classes and other sets, is called an _abstract syntax definition_. As an example, the following abstract syntax definition specifies a set of data structures which might be used to represent algebraic expressions involving addition and multiplication:

```
        (CLASS, CONSTANT, (VALUE, INTCLASS)),
        (CLASS, VARIABLE, (STRING, SEQ, CHARCLASS)),
        (CLASS, SUM, (LEFT, EXP), (RIGHT, EXP)),
        (CLASS, PRODUCT, (LEFT, EXP), (RIGHT, EXP)),
        (UNION, EXP, CONSTANT, VARIABLE, SUM, PRODUCT)
```

Now suppose that d_1, ... , d_n is an abstract syntax definition, and that each identifier in this definition has been declared to denote a distinct atom. Then

D IS (d_1, ... , d_n);

is a GEDANKEN declaration binding D to a data structure which is an encoding of the abstract syntax definition. We now proceed to define GEDANKEN functions, using the value of D, for creating records and testing set membership.

As a preliminary, we define the following generally useful functions for manipulating sequences:

```
CONC IS λ(X, Y) VECTOR(1, ADD(X UL, Y UL),
    λI IF GREATER(I, X UL) THEN Y SUBTRACT(I, X UL) ELSE X I);
CONS IS λ(X, Y) CONC(UNITSEQ X, Y);
AUG IS λ(X, Y) CONC(X, UNITSEQ Y);
SUBSEQ IS λ(L, U, X) VECTOR(1, INC SUBTRACT(U, L), λI X DEC ADD(I, L));
HEAD IS λX SUBSEQ(1, DEC X UL, X);
TAIL IS λX SUBSEQ(2, X UL, X);
REPLACE IS λ(I, V, X) VECTOR(1, X UL, λJ IF J = I THEN V ELSE X J);
SEARCH ISR λ(N, P, F, G)
    IF NOT GREATER(N, 0) THEN G() ELSE IF P N THEN F N ELSE SEARCH(DEC N, P, F, G);
STREQUAL ISR λ(X, Y)
    X UL = Y UL AND SEARCH(X UL, λI NOT(X I = Y I), λI FALSE, λ() TRUE);
```

The function CONC concatenates two sequences. CONS (or AUG) adds a component to the beginning (or end) of a sequence. SUBSEQ(L, U, X) produces the subsequence of X which begins with the Lth component and ends with the Uth component. HEAD (or TAIL) reproduces its argument with the last (or first) component deleted. REPLACE(I, V, X) reproduces the sequence X with the Ith component replaced by V.

The function SEARCH(N, P, F, G) is used for searching through a sequence. It successively tests the predicate P I for the integers I = N, N-1, ... , 1. If P I is true, it evaluates F I and returns the result. If all tests of P are false, it evaluates G () and returns the result. The function STREQUAL uses SEARCH to determine if two sequences of primitive data are equal.

In the sequel, we will frequently use sequences which obey a stack discipline. In such cases, the first sequence component will be the most recently added stack element, so that stack elements will be added by using the function CONS and deleted by TAIL.

We now define functions which use the value of D to test and create records. The following functions test set membership:

```
T ISR λ(X, S)
    IF S = INTCLASS THEN ISINTEGER X ELSE IF S = BOOLCLASS THEN ISBOOLEAN X
    ELSE IF S = CHARCLASS THEN ISCHAR X ELSE IF S = UNIVERSAL THEN TRUE
    ELSE SEARCH(D UL, λI (D I ) 2 = S,
        λI IF (D I) 1 = CLASS THEN ISFUNCTION X AND X TYPE = S
            ELSE TUNION(X, TAIL TAIL D I),
        λ() GOTO ERROR);
TUNION ISR λ(X, U)
    SEARCH(U UL, λI T(X, U I), λI TRUE, λ() FALSE);
TSEQ ISR λ(X, S)
    X LL = 1 AND SEARCH(X UL, λI NOT T(X I, S), λI FALSE, λ() TRUE);
```

T(X, S) accepts a record or primitive datum X and a set name S, and tests whether X is a member of the set denoted by S. TUNION(X, U) accepts a sequence U of set names, and tests whether the record or primitive datum X is a member of any of the sets denoted by the components of U. TSEQ(X, S) accepts a vector X whose components are records or primitive data, and tests whether X is a sequence in which every component is a member of the set denoted by the set name S.

The following function creates records:

```
M ISR λX
    (C, V IS IF ISATOM X THEN X, () ELSE (X 1, TAIL X);
    SEARCH(D UL, λI (D I) 1 = CLASS AND (D I) 2 = C,
        λI M1(C, V, TAIL TAIL D I), λ() GOTO ERROR));
    M1 ISR λ(C, V, F)
        IF V UL = F UL AND SEARCH(V UL,
           λI NOT(IF (F I) 2 = SEQ THEN TSEQ(V I, (F I) 3) ELSE T(V I, (F I) 2)),
           λI FALSE, λ() TRUE)
        THEN λX IF X = TYPE THEN C ELSE SEARCH(V UL, λI X = (F I) 1, V,
           λ() GOTO ERROR)
        ELSE GOTO ERROR;
```

M accepts an argument (C, v_1, ... , v_n), where C is a class name and the v_i are field values, which may be primitive data, records, or sequences thereof. The special case n = 0 (corresponding to a record with no fields) is treated differently, so that one may write M C instead of M UNITSEQ C. If C has the class definition (CLASS, C, (f_1, r_1), ... , (f_n, r_n)) then M produces a record R in class C such that R f_i = v_i. Checks are performed to insure that the number of field values is correct, and that each v_i satisfies the range descriptor r_i.

In manipulating records, we will use the functions T(X, S) and M(C, v_1, ... , v_n) only in function designators in which S and C are constant atoms. There is a serious inefficiency in this situation, since each time a particular function designator is executed it will perform a search over the value of D whose outcome will always be the same. This inefficiency is irrelevant to our present purpose of using GEDANKEN as a vehicle for formal definition, but it is symptomatic of a limitation of the language, which might be overcome by introducing macro-definitional facilities.

As a simple example of the use of T and M, the following function will produce the (unsimplified) derivative of an expression X by a variable Y, using the abstract syntax for algebraic expressions given above:

```
DERV ISR λ(X, Y)
    IF T(X, CONSTANT) THEN M(CONSTANT, 0)
    ELSE IF T(X, VARIABLE) THEN M(CONSTANT,
       IF STREQUAL(X STRING, Y STRING) THEN 1 ELSE 0)
    ELSE IF T(X, SUM) THEN M(SUM, DERV(X LEFT, Y), DERV(X RIGHT, Y))
    ELSE M(SUM, M(PRODUCT, DERV(X LEFT, Y), X RIGHT),
       M(PRODUCT, X LEFT, DERV(X RIGHT, Y)));
```

Abstract Syntax of GEDANKEN

We now give an abstract syntax definition specifying the set of GEDANKEN data structures which will be used to represent GEDANKEN programs. It is evident that this abstract syntax will be considerably simpler than the concrete (BNF) syntax of GEDANKEN, e.g., the distinction between different precedence levels for expressions and parameter forms will disappear. Further simplification is obtained by deleting various language constructions which can be treated as abbreviations:

```
(UNION, EXP, CONSTANT, IDENT, FUNCTDES, LAMBDAEXP, CONDEXP, CASEEXP, BLOCK),
(CLASS, CONSTANT, (VALUE, VALUEDEN)),
(UNION, IDENT, PROGIDENT, INTRIDENT),
(CLASS, PROGIDENT, (STRING, SEQ, CHARCLASS)),
(CLASS, INTRIDENT, (NAME, INTCLASS)),
(CLASS, FUNCTDES, (FUNCTPART, EXP), (ARGPART, EXP)),
(CLASS, LAMBDAEXP, (PARAMPART, IDENT), (BODY, EXP)),
(CLASS, CONDEXP, (PREMISS, EXP), (CONCLUSION, EXP), (ALTERNATIVE, EXP)),
(CLASS, CASEEXP, (INDEX, EXP), (BODY, SEQ, EXP)),
(CLASS, BLOCK, (RDECLPART, SEQ, RDECL), (LDECLPART, SEQ, LDECL),
   (BODY, SEQ, EXP)),
(CLASS, RDECL, (LEFT, IDENT), (RIGHT, LAMBDAEXP)),
(CLASS, LDECL, (LEFT, IDENT), (RIGHT, SEQ, EXP)),
(UNION, PFORM, IDENT, SEQPFORM),
(CLASS, SEQPFORM, (BODY, SEQ, PFORM))
```

The following comments pertain to the above definition:

(1) We postpone defining the set named VALUEDEN (value denotations) until the discussion of the interpreter. In general, value denotations will be data structures used in the interpreter to represent values computed by the program which is being interpreted. For the moment, we only specify that there is a distinct value denotation for every member of the universal value set, and that the set of value denotations includes integers, boolean values, and characters, which all denote themselves.

(2) A variety of forms occurring in the concrete syntax do not occur in the abstract syntax, e.g., expressions involving =, AND, OR, or :=, sequence expressions, sequence parameter forms, and nonrecursive declarations. Each of these forms can be regarded as an abbreviation, and is eliminated during the conversion of a concrete program into an abstract program. (Note that sequence parameter forms are still defined in the abstract syntax, but are not allowed to occur in any type of expression. This reflects the fact that these forms will be used as temporary quantities during the concrete-abstract conversion, but will not occur in the final result of this conversion.)

(3) Abstract programs will contain two classes of identifiers: program identifiers (PROGIDENT), which are images of identifiers occurring in the original concrete program; and internal identifiers (INTRIDENT), which are introduced during the conversion process. Different internal identifiers are distinguished by their integer-valued NAME fields.

(4) The implicit declaration of labels is made explicit in the abstract syntax. Labels no longer appear in the body of a block, which is simply a sequence of unlabelled statements (i.e., expressions). Instead, each identifier used as a label occurs in a label declaration, in which it is paired with the sequence of unlabelled statements to be executed after a jump to the label has occurred.

Concrete Syntax of GEDANKEN

The syntax of concrete programs is defined in two stages: (1) A unique partitioning of the program into a sequence of character strings called tokens is specified, and then (2) The set of well-formed programs is defined by a grammar over the infinite vocabulary of tokens.

The tokens themselves satisfy the following grammar:

```
<character> ::= " | <quotable character>
<quotable character> ::= <letter or digit> | λ | , | = | : | ( | ) | ; |
     ␣ | <extra character>
<letter or digit> ::= <letter> | <digit>
<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
     P | Q | R | S | T | U | V | W | X | Y | Z
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<token> ::= <integer token> | <quoted string token> | <word token> |
     <punctuation token>
<integer token> ::= <digit> { <digit>}*
<quoted string token> ::= "{<quotable character>}*"
<word token> ::= <letter> { <letter or digit>}*
<punctuation token> ::= λ | , | = | : | ( | ) | ; | :=
```

Here the symbol ␣ denotes a blank and the undefined class <extra character> denotes the set of all input-representable characters not occurring elsewhere in the syntax.

A concrete GEDANKEN program must be a sequence of tokens separated by zero or more blanks, with at least one blank used to separate any of the following pairs:

Left Token	Right Token
<integer token>	<integer token>
<word token>	<integer token>
<word token>	<word token>
:	=

This separation condition is sufficient to insure the unique partitioning of any program.

The class of <word token>'s is subdivided into <reserved word token>'s, which are the strings AND, OR, IF, THEN, ELSE, CASE, OF, IS, and ISR, and <identifier token>'s, which are all other <word token>'s.

Once a program has been partitioned into tokens, and the separating blanks have been deleted, it is parsed according to the grammar given on the lefthand side of Table I. (This grammar is equivalent to that given in Chapter II, but has been altered somewhat to simplify the conversion to abstract syntax.) In the productions of this grammar, reserved word and punctuation tokens appear as terminal objects, while identifier, integer, and quoted string tokens appear as undefined syntactic classes.

Parsing methods are well-understood, and will not be described here. We simply assert that a well-formed GEDANKEN program can be transformed into a <u>derivation tree</u> with the following properties:

(1) Each node is associated with either a token or a production of the grammar in Table I.

(2) A node associated with a token is a terminal node.

Productions	Associated Translation Functions
<identifier> ::= <identifier token>	λX M(PROGIDENT, X)
<exp$_0$> ::= <integer token>	λX M(CONSTANT, CONVERTINT X)
| <quoted string token>	λX TRANSTRING HEAD TAIL X
| <identifier>	λX X
| (<block$_2$>)	λX X
<exp$_1$> ::= <exp$_0$>	λX X
| <exp$_0$><exp$_1$>	λ(X, Y) M(FUNCTDES, M(FUNCTDES, COERCECON, X), Y)
<exp$_2$> ::= <exp$_1$>	λX X
| <exp$_1$> = <exp$_2$>	λ(X, Y) M(FUNCTDES, EQUALCON, TRANSEQEXP(X, Y))
<exp$_3$> ::= <exp$_2$>	λX X
| <exp$_2$> AND <exp$_3$>	λ(X, Y) M(CONDEXP, M(FUNCTDES, COERCECON, X), M(FUNCTDES, COERCECON, Y), M(CONSTANT, FALSE))
exp$_4$> ::= <exp$_3$>	λX X
| <exp$_3$> OR <exp$_4$>	λ(X, Y) M(CONDEXP, M(FUNCTDES, COERCEON, X), M(CONSTANT, TRUE), M(FUNCTDES, COERCECON, Y))
:exp$_5$> ::= <exp$_4$>	λX X
| IF <exp$_6$> THEN <exp$_6$> ELSE <exp$_5$>	λ(X, Y, Z) M(CONDEXP, M(FUNCTDES, COERCECON, X), Y, Z)
| λ <pform$_0$><exp$_5$>	TRANSLAMBDA
| <exp$_4$> := <exp$_5$>	λ(X, Y) M(FUNCTDES, SETCON, TRANSEQEXP(X, Y))

TABLE I.

Productions	Associated Translation Functions
<exp$_6$> ::= <exp$_5$>	λX X
| <empty>	TRANSEQEXP
| <exp$_5$> , <exp$_5$> {, <exp$_5$>}*	TRANSEQEXP
| CASE <exp$_6$> OF <exp$_5$> {, <exp$_5$>}*	λX M(CASEEXP, M(FUNCTDES, COERCECON, X 1), TAIL X)
<pform$_0$> ::= <identifier>	λX X
| (<pform$_1$>)	λX X
<pform$_1$> ::= <pform$_0$>	λX X
| <empty>	λX M(SEQPFORM, X)
| <pform$_0$> , <pform$_0$> {, <pform$_0$>}*	λX M(SEQPFORM, X)
<block$_0$> ::= <exp$_6$>	λX M(BLOCK, (), (), UNITSEQ X)
| <exp$_6$> ; <block$_0$>	λ(X, Y) M(BLOCK, (), Y LDECLPART, CONS(X, Y BODY))
| <identifier> : <block$_0$>	λ(X, Y) M(BLOCK, (), CONS(M(LDECL, X, Y BODY), Y LDECLPART), Y BODY)
<block$_1$> ::= <block$_0$>	λX X
| <identifier> ISR λ <pform$_0$><exp$_5$>; <block$_1$>	λ(X, Y, Z, W) M(BLOCK, CONS(M(RDECL, X, TRANSLAMBDA(Y, Z)), W RDECLPART), W LDECLPART, W BODY)
<block$_2$> ::= <block$_1$>	λX X
| <pform$_1$> IS <exp$_6$>; <block$_2$>	TRANSDECL
<program> ::= <block$_2$>	λX X

TABLE I (continued)

(3) If a node is associated with a production which has n items on its right side, then the node has n subnodes. If the ith item on the right side of the production is a specific token (token class, syntactic class), then the ith subnode is associated with the specific token (some member of the token class, some production whose left side is the syntactic class).

Translation into Abstract Form

The conversion of a derivation tree into an abstract program is an instance of syntax-directed translation.(14,9) With each production of the grammar in Table I, we associate a GEDANKEN function which expresses the translation of any phrase which is an instance of that production in terms of the translations of its subphrases.

This process can be described more precisely as follows: The nodes of the derivation tree are translated in some order such that no node is translated until after all of its subnodes have been translated. If a node is associated with a token, its translation is a sequence whose components are the characters of the token. If a node is associated with a production, its translation is obtained by applying the corresponding translation function to a sequence whose components are the translations of the immediate subnodes, excepting those subnodes which are associated with reserved word or punctuation tokens. In the special case where this sequence has a single component, the component itself, rather than the one-component sequence, is used as the argument to the translation function.

A compound production containing the metasymbol | is regarded as an abbreviation for a set of productions with the same left side; each of these productions may have a distinct associated translation function. A production containing the metasymbols { }* is regarded as an abbreviation for the infinite set of productions whose members are formed by replicating the bracketed elements k times, for each k $\geq$ 0; all of these productions will have the same associated translation function. When the right side of a production is <empty>, the associated translation function receives an empty sequence as its argument. The following subsidiary functions and other values are used by the translation functions:

```
    IICOUNT IS REF 0;
    CRIDENT IS λ() (IICOUNT := INC IICOUNT; M(INTRIDENT, VAL IICOUNT));
    EQUALCON IS M(CONSTANT, GETVALPREDEF M(PROGIDENT, "EQUAL"));
    SETCON IS M(CONSTANT, GETVALPREDEF M(PROGIDENT, "SET"));
    COERCECON IS M(CONSTANT, GETVALPREDEF M(PROGIDENT, "COERCE"));
    CONVERTINT ISR λX IF X UL = 0 THEN 0 ELSE
        ADD(DIGITTOINT X X UL, MULTIPLY(10, CONVERTINT HEAD X));
    TRANSTRING IS λX IF X UL = 1 THEN M(CONSTANT, X 1) ELSE
        (I IS CRIDENT(); M(LAMBDAEXP, I, M(CASEEXP, M(FUNCTDES, COERCECON, I),
            VECTOR(1, X UL, λJ M(CONSTANT, X J)))));
    TRANSDECL ISR λ(P, E, B) M(FUNCTDES, TRANSLAMBDA(P, B), E);
    TRANSLAMBDA ISR λ(P, B) IF T(P, IDENT) THEN M(LAMBDAEXP, P, B)
        ELSE (I IS CRIDENT(); K IS REF (P BODY) UL; R IS REF B;
        LOOP: IF K = 0 THEN GOTO DONE ELSE
            R := TRANSDECL((P BODY) VAL K, M(FUNCTDES,
               M(FUNCTDES, COERCECON, I), M(CONSTANT, VAL K)), VAL R);
            K := DEC K; GOTO LOOP;
        DONE: M(LAMBDAEXP, I, VAL R));
    TRANSEQEXP ISR λX
        (S IS VECTOR(1, X UL, λJ CRIDENT()); I IS CRIDENT(); K IS REF X UL;
        R IS REF M(LAMBDAEXP, I, M(CASEEXP, M(FUNCTDES, COERCECON, I), S));
```

```
LOOP: IF K = 0 THEN GOTO DONE ELSE
    R := TRANSDECL(S VAL K, X VAL K, VAL R);
    K := DEC K; GOTO LOOP;
DONE: VAL R);
```

The global reference IICOUNT maintains a count of the number of internal identifiers which have been created during the translation process. It is used by the function CRIDENT(), which returns a distinct internal identifier each time it is executed.

The values of EQUALCON, SETCON, and COERCECON are constants denoting the basic functions EQUAL, SET, and COERCE. The value fields of these constants are obtained from the function GETVALPREDEF (to be defined later), which produces the value denotations of predefined identifiers. The use of these constants instead of the corresponding identifiers insures that redeclaration of the identifiers will not affect implicit coercion or the meaning of the operations = and :=.

CONVERTINT converts a sequence of digits into the corresponding integer. TRANSTRING translates quoted string tokens (after deletion of the enclosing quote characters). If $n \neq 1$, the token "$c_1 \ldots c_n$" is translated into the abstract equivalent of λi (CASE i OF "c_1", ... , "c_n").

The three interconnected functions TRANSDECL, TRANSLAMBDA, AND TRANSEQEXP eliminate declarations, sequence parameter forms, and sequence expressions. Their effect is essentially equivalent to repeated application of the following equivalences:

p IS e; b $\Rightarrow$ (λ(p)(b))(e)

λ(p_1, ... , p_n) b   (when $n \neq 1$)

  $\Rightarrow \lambda$i (p_1 IS i 1; ... ; p_n IS i n'; b)

e_1, ... , e_n (when $n \neq 1$)

$\Rightarrow$(i_1 IS e_1; ... ; i_n IS e_n; λi (CASE i OF i_1, ... , i_n))

where n' is an integer constant whose value is n, and i, i_1, ... , i_n are unique internal identifiers.

A final effect of the translation process is to insert explicit calls of COERCE for the implicit coercion performed by AND-, OR-, conditional, and case expressions, function designators, sequence parameter forms, and the functions produced by sequence expressions.

Semi-Basic Functions

Some of the basic functions in GEDANKEN have been introduced for reasons of convenience or efficiency, and are not basic in a theoretical sense. Rather than including these functions in the abstract language accepted by our interpreter function, we will eliminate them by defining them in terms of lambda-expressions involving the remaining basic functions.

Thus we assume that a concrete GEDANKEN program, before being parsed, will be enclosed in parentheses and preceded by the following standard declarations:

```
UNITSEQ IS λX λI (CASE I OF X);
NOT IS  λX IF X THEN FALSE ELSE TRUE;
INTTODIGIT IS λX
    (CASE INC X OF "0", "1", "2", "3", "4", "5", "6", "7", "8", "9");
DIGITTOINT IS λX (X IS COERCE X;
    IF X = "0" THEN 0 ELSE IF X = "1" THEN 1 ELSE IF X = "2" THEN 2 ELSE
    IF X = "3" THEN 3 ELSE IF X = "4" THEN 4 ELSE IF X = "5" THEN 5 ELSE
    IF X = "6" THEN 6 ELSE IF X = "7" THEN 7 ELSE IF X = "8" THEN 8 ELSE
    IF X = "9" THEN 9 ELSE GOTO ERROR);
VECTOR ISR λ(L, U, F) (L IS COERCE L; U IS COERCE U; F IS COERCE F;
    IF GREATER(L, U) THEN λI (I IS COERCE I;
       IF I = LL THEN L ELSE IF I = UL THEN DEC L ELSE GOTO ERROR)
    ELSE (V IS VECTOR(L, DEC U, F); T IS F U; λI (I IS COERCE I;
       IF I = UL THEN U ELSE IF I = U THEN T ELSE V I)));
```

```
NEG ISR λX (X IS COERCE X; IF NOT ISINTEGER X THEN GOTO ERROR
    ELSE IF X = 0 THEN 0 ELSE IF GREATER(X, 0) THEN DEC NEG DEC X
    ELSE INC NEG INC X);
ADD ISR λ(X, Y) (X IS COERCE X; Y IS COERCE Y;
    IF NOT ISINTEGER X OR NOT ISINTEGER Y THEN GOTO ERROR
    ELSE IF X = 0 THEN Y ELSE IF GREATER(X, 0) THEN INC ADD(DEC X, Y)
    ELSE DEC ADD(INC X, Y));
SUBTRACT ISR λ(X, Y) (X IS COERCE X; Y IS COERCE Y; ADD(X, NEG Y));
MULTIPLY ISR λ(X, Y) (X IS COERCE X; Y IS COERCE Y;
    IF NOT ISINTEGER X OR NOT ISINTEGER Y THEN GOTO ERROR
    ELSE IF X = 0 THEN 0 ELSE IF GREATER(X, 0) THEN ADD(MULTIPLY(DEC X, Y), Y)
    ELSE SUBTRACT(MULTIPLY(INC X, Y), Y));
DIVIDE ISR λ(X, Y) (X IS COERCE X; Y IS COERCE Y;
    IF NOT ISINTEGER X OR NOT ISINTEGER Y OR Y = 0 THEN GOTO ERROR
    ELSE IF GREATER(0, Y) THEN NEG(DIVIDE(X, NEG Y))
    ELSE IF NOT GREATER(Y, X) THEN INC DIVIDE(SUBTRACT(X, Y), Y)
    ELSE IF NOT GREATER(Y, NEG X) THEN DEC DIVIDE(ADD(X, Y), Y) ELSE 0);
REMAINDER ISR λ(X, Y) (X IS COERCE X; Y IS COERCE Y;
    SUBTRACT(X, MULTIPLY(Y, DIVIDE(X, Y))));
```

For the remainder of this chapter, we will no longer consider the above-defined functions to be basic.

The Interpreter

We now develop the definition of a function, called INTERPRET, which accepts an abstract program (i.e., a data structure belonging to the set named EXP) and produces (the denotation of) its value. Essentially, this function simulates a machine which passes through a succession of <u>states</u>. Each state is obtained from the preceding state by applying the function TRANSITION. Certain terminal states, which satisfy the predicate ISTERMSTATE, cause the machine to stop and return a value. Thus we have:

```
     INTERPRET ISR λP INTERPRET1 INITIALSTATE P;
     INTERPRET1 ISR λS IF ISTERMSTATE S THEN FINALVALUE S
          ELSE INTERPRET1 TRANSITION S;
```

where S is always bound to a record in the class named STATE.

Before defining the subsidiary functions of INTERPRET, we must specify the abstract syntax of _states_. A STATE is a record with six fields:

```
     (CLASS, STATE, (CONTROL, SEQ, INST), (STACK, SEQ, VALUEDEN),
          (ENVR, SEQ, ENVELEM), (DUMP, SEQ, DUMPELEM),
          (MEMORY, SEQ, VALUEDEN), (ATOMCOUNT, INTCLASS))
```

The _control_ is a sequence of _instructions_ to be executed. They are similar to the instructions of a conventional computer, except that certain instructions, when executed, may expand into several simpler instructions instead of being deleted from the sequence. The _stack_ is a sequence of value denotations which is used to store intermediate results during the evaluation of compound expressions. The _environment_ is a sequence of _environment elements_, each of which specifies the current binding of an identifier (except for environment marks, which are described below).

The _dump_ is a sequence of _dump elements_, with the syntax

```
     (CLASS, DUMPELEM, (CONTROL, SEQ, INST), (STACK, SEQ, VALUEDEN),
          (ENVR, SEQ, ENVELEM)),
```

which are used to save the control, stack, and environment portions of the state whenever a new block or lambda-expression body is evaluated. A new element is added to the dump whenever such evaluation begins, and the appropriate fields of the state are restored from this element when the evaluation is completed.

The memory is a sequence of value denotations which specifies the values of references. Each (nonimplicit) reference denotation will contain an integer-valued field named ADDRESS; if the value of this field is i, then the value of the denoted reference will be the ith component of the memory. The atom count is an integer giving the total number of atoms which have been created during the execution of the program.

Value denotations have the following abstract syntax:

```
(UNION, VALUEDEN, INTCLASS, BOOLCLASS, CHARCLASS, ATOMDEN, FUNCTDEN,
    REFDEN, LABELDEN),
(UNION, ATOMDEN, LLDEN, ULDEN, PROGATOMDEN),
(CLASS, LLDEN),
(CLASS, ULDEN),
(CLASS, PROGATOMDEN, (NAME, INTCLASS)),
(CLASS, FUNCTDEN, (CONTROL, SEQ, INST), (ENVR, SEQ, ENVELEM)),
(UNION, REFDEN, EXPREFDEN, IMPREFDEN),
(CLASS, EXPREFDEN, (ADDRESS, INTCLASS)),
(CLASS, IMPREFDEN, (SETF, FUNCTDEN), (VALF, FUNCTDEN)),
(UNION, LABELDEN, ERRORDEN, PROGLABELDEN),
(CLASS, ERRORDEN),
(CLASS, PROGLABELDEN, (BODY, SEQ, EXP), (ENVR, SEQ, ENVELEM),
    (DUMP, SEQ, DUMPELEM))
```

Fieldless record classes, such as LLDEN and ULDEN, each contain a single record which is only applicable to the argument TYPE. The records in LLDEN and ULDEN denote the atoms LL and UL. All other atoms are denoted by records in the class PROGATOMDEN, and are distinguished by the integer values of their name fields. Records in the class FUNCTDEN denote functions; their CONTROL field gives a sequence of instructions for evaluating the function, and their ENVR field gives an initial environment to be used during the execution of these instructions.

Explicit references are denoted by the class EXPREFDEN; the integer-valued ADDRESS field distinguishes distinct references and specifies the component of the memory which is the value of the reference. Implicit references are denoted by the class IMPREFDEN; the fields SETF and VALF give denotations of the functions for setting and evaluating the reference.

The single record in the class ERRORDEN denotes the label ERROR. All other labels are denoted by the class PROGLABELDEN, in which the field BODY gives a sequence of expressions to be executed when a jump to the label occurs, ENVR gives the identifier bindings for these expressions, and DUMP gives a sequence of dump elements indicating the computation to be carried out after the expressions in BODY have been evaluated. The values of ENVR and DUMP constitute the "current status" discussed in the previous chapter.

We next consider the syntax of environments. An <u>environment element</u> is normally a pair, consisting of an identifier and the denotation of the value to which it is bound. Thus, except for the difficulties discussed below, we <u>would</u> have the syntax:

    (CLASS, ENVELEM, (LEFT, IDENT), (RIGHT, VALUEDEN))

But this form of environment is incapable of describing the effects of recursive declarations and label declarations. These declarations must bind identifiers to function and label denotations whose ENVR field specifies an environment containing the new binding. Unfortunately, this type of explicit circularity cannot be represented by abstract syntactic data structures (which cannot be implicit data structures nor contain imbedded references).

To avoid this difficulty, we use the following subterfuge: recursive declarations and label declarations will bind identifiers to special <u>recursive</u> denotations which do not contain ENVR fields. During the execution of each block, immediately after all declarations have been executed, an element called an <u>environment mark</u> will be added to the environment. Then, whenever a search of the environment yields a recursive denotation, an ENVR field will be added to the denotation which specifies the portion of the searched environment beginning at the last-encountered mark.

Thus <u>environment elements</u> have the following syntax:

```
    (UNION, ENVELEM, ENVMARK, ENVPAIR),
    (CLASS, ENVMARK),
    (CLASS, ENVPAIR, (LEFT, IDENT), (RIGHT, ENVVALUEDEN)),
    (UNION, ENVVALUEDEN, VALUEDEN, RECFUNCTDEN, RECLABELDEN),
    (CLASS, RECFUNCTDEN, (CONTROL, SEQ, INST)),
    (CLASS, RECLABELDEN, (BODY, SEQ, EXP), (DUMP, SEQ, DUMPELEM))
```

The function GETVAL(X, E) searches an environment E (in increasing order of components) to obtain the denotation of the value bound to an identifier X:

```
    GETVAL ISR λ(X, E) GETVAL1(X, E, ());
    GETVAL1 ISR λ(X, E, EM)
        IF E UL = 0 THEN GETVALPREDEF X
        ELSE IF T(E 1, ENVMARK) THEN GETVAL1(X, TAIL E, E)
        ELSE IF IDEQUAL(X, (E 1) LEFT) THEN
            (V IS (E 1) RIGHT;
            IF T(V, RECFUNCTDEN) THEN M(FUNCTDEN, V CONTROL, EM)
            ELSE IF T(V, RECLABELDEN) THEN M(PROGLABELDEN, V BODY, EM, V DUMP)
            ELSE V)
        ELSE GETVAL1(X, TAIL E, EM);
    IDEQUAL ISR λ(X, Y)
        T(X, PROGIDENT) AND T(Y, PROGIDENT) AND STREQUAL(X STRING, Y STRING)
        OR T(X, INTRIDENT) AND T(Y, INTRIDENT) AND X NAME = Y NAME;
    GETVALPREDEF ISR λX
        SEARCH(30, λI T(X, PROGIDENT) AND STREQUAL(X STRING, PREDEFIDSTRS I),
            PREDEFVDENS, λ() GOTO ERROR);
```

The subsidiary function IDEQUAL determines if two identifiers are equal. The function GETVALPREDEF, which produces the value denotations of predefined identifiers, uses two parallel sequences PREDEFIDSTRS and PREDEFVDENS, whose components are the identifier string fields and value denotations of the predefined identifiers. The values of these sequences will be given later.

The final set for which we must give an abstract syntax is the set of <u>instructions</u>:

```
(UNION, INST, EXP, RDECL, LDECL, EXEC, BRANCH, SELECT, BIND, APPLY,
    MARKENV, DELETE, BASICFUNCTINST),
(CLASS, EXEC, (BODY, SEQ, EXP)),
(CLASS, BRANCH, (CONCLUSION, EXP), (ALTERNATIVE, EXP)),
(CLASS, SELECT, (BODY, SEQ, EXP)),
(CLASS, BIND, (BODY, IDENT)),
(CLASS, APPLY),
(CLASS, MARKENV),
(CLASS, DELETE),
(CLASS, BASICFUNCTINST, (STRING, SEQ, CHARCLASS))
```

The following is an informal description of the effect of each class of instructions. Certain instructions, such as compound expressions, require repeated applications of TRANSITION to be completed; a single application of TRANSITION will cause such an instruction to be replaced by a sequence of simpler instructions.

EXP: The expression is evaluated and its value is added to the stack.

RDECL: A recursive function denotation is created from the lambda expression on the right of the declaration, and the environment is extended by binding the identifier on the left to this denotation.

LDECL: A recursive label denotation is created from the expression sequence on the right of the declaration and the current value of the dump, and then the environment is extended by binding the identifier on the left to this denotation.

EXEC: The sequence of expressions in the instruction body is evaluated, and the value of the last expression is added to the stack.

BRANCH: If the first stack component is TRUE (or FALSE), then the CONCLUSION (or ALTERNATIVE) is evaluated and its value replaces the first stack component.

SELECT: If the first stack component is an integer i, then the ith component of the instruction body is evaluated and its value replaces the first stack component. If the first stack component is the atom LL (or UL), then it is replaced by 1 (or the length of the instruction body).

BIND: The environment is extended by binding the identifier in the instruction body to the first stack component, which is deleted from the stack.

APPLY: The second stack component, which must be a function denotation, is applied to the first stack component, and the result of this application replaces the first two stack components.

MARKENV: A mark is added to the environment.

DELETE: The first stack component is deleted.

BASICFUNCTINST: A basic function instruction denotes the basic function whose predefined identifier has the same STRING field. Prior to the execution of the basic function instruction, the following actions will have occurred: (1) The arguments will be "spread," i.e., if the function expects a sequence of n arguments, the components of this sequence will be placed in the first n stack positions (with the last component in the first stack position).

(2) The arguments will have been coerced appropriately. Then the effect of the basic function instruction is to evaluate the corresponding basic function and replace the first n stack components with its result.

We may now define the subsidiary functions of INTERPRET. In the initial state, the control contains the program to be interpreted, the atom count is zero, and the remaining state fields are empty. A state is terminal if its control and dump are both empty; the resulting value is the first (and only) component of the stack. Thus we have:

```
INITIALSTATE IS λP M(STATE, UNITSEQ P, (), (), (), (), 0);
ISTERMSTATE IS λS (S CONTROL) UL = 0 AND (S DUMP) UL = 0;
FINALVALUE IS λS (S STACK) 1;
```

The heart of the interpreter is the function TRANSITION. If the control is empty, TRANSITION replenishes the control, stack, and environment from the dump, saving the first component of the stack, which will be the value of the block or function designator whose evaluation has just been completed. Otherwise, a massive branch on the class of the current instruction is executed, and various fields of the state are updated in accordance with this instruction.

Since a particular instruction will normally affect only a few state fields, it is inconvenient to show the unchanged fields explicitly for each instruction. Thus we proceed in the following manner: (1) Before the instruction branch, references are created to each field value (after deleting the current instruction from the control field). (2) Each instruction resets the references to the fields which are updated. (3) After the branch, a new state is constructed from the reference values.

TRANSITION is defined so that it will always terminate, even when used to interpret a nonterminating GEDANKEN program. Such programs will cause the interpreter to repeatedly apply TRANSITION without ever obtaining a terminal state.

```
TRANSITION IS λS
(STV, ENV, DMV, MRV, ACV IS S STACK, S ENVR, S DUMP, S MEMORY, S ATOMCOUNT;
IF (S CONTROL) UL = 0 THEN
    M(STATE, (DMV 1) CONTROL, CONS(STV 1, (DMV 1) STACK),
        (DMV 1) ENVR, TAIL DMV, MRV, ACV)
ELSE
(X IS (S CONTROL) 1; CNV IS TAIL S CONTROL;
CN, ST, EN, DM, MR, AC IS REF CNV, REF STV, REF ENV, REF DMV, REF MRV, REF ACV;
S1, T1, S2, T2 IS IF STV UL = 0 THEN (), (), (), ()
    ELSE IF STV UL = 1 THEN STV 1, TAIL STV, (), ()
    ELSE STV 1, TAIL STV, STV 2, TAIL TAIL STV;
IF T(X, CONSTANT) THEN ST := CONS(X VALUE, STV)
ELSE IF T(X, IDENT) THEN ST := CONS(GETVAL(X, ENV), STV)
ELSE IF T(X, FUNCTDES) THEN CN := CONC((X FUNCTPART, X ARGPART, M APPLY), CNV)
ELSE IF T(X, LAMBDAEXP) THEN
    ST := CONS(M(FUNCTDEN, (M(BIND, X PARAMPART), X BODY), ENV), STV)
ELSE IF T(X, CONDEXP) THEN
    CN := CONC((X PREMISS, M(BRANCH, X CONCLUSION, X ALTERNATIVE)), CNV)
ELSE IF T(X, CASEEXP) THEN CN := CONC((X INDEX, M(SELECT, X BODY)), CNV)
ELSE IF T(X, BLOCK) THEN
    (DM := CONS(M(DUMPELEM, CNV, STV, ENV), DMV); ST := ();
    CN := CONC(X RDECLPART, CONC(X LDECLPART, (M MARKENV, M(EXEC, X BODY)))))
ELSE IF T(X, RDECL) THEN
    EN := CONS(M(ENVPAIR, X LEFT,
        M(RECFUNCTDEN, (M(BIND, (X RIGHT) PARAMPART), (X RIGHT) BODY))), ENV)
ELSE IF T(X, LDECL) THEN
    EN := CONS(M(ENVPAIR, X LEFT, M(RECLABELDEN, X RIGHT, DMV)), ENV)
```

```
ELSE IF T(X, EXEC) THEN
    (B IS X BODY; CN := IF B UL = 1 THEN CONS(B 1, CNV)
       ELSE CONC((B 1, M DELETE, M(EXEC, TAIL B)), CNV))
ELSE IF T(X, BRANCH) THEN
    IF T(S1, BOOLCLASS) THEN
       (CN := CONS(X (IF S1 THEN CONCLUSION ELSE ALTERNATIVE), CNV);
       ST := T1)
    ELSE GOTO ERROR
ELSE IF T(X, SELECT) THEN
    IF T(S1, INTCLASS) AND NOT GREATER(1, S1) AND NOT GREATER(S1, (X BODY) UL)
       THEN (CN := CONS((X BODY) S1, CNV); ST := T1)
    ELSE IF T(S1, LLDEN) THEN ST := CONS(1, T1)
    ELSE IF T(S1, ULDEN) THEN ST := CONS((X BODY) UL, T1)
    ELSE GOTO ERROR
ELSE IF T(X, BIND) THEN
    (EN := CONS(M(ENVPAIR, X BODY, S1), ENV); ST := T1)
ELSE IF T(X, APPLY) THEN
    IF T(S2, FUNCTDEN) THEN
       (DM := CONS(M(DUMPELEM, CNV, T2, ENV), DMV);
       CN := S2 CONTROL; EN := S2 ENVR; ST := UNITSEQ S1)
    ELSE GOTO ERROR
ELSE IF T(X, MARKENV) THEN EN := CONS(M ENVMARK, ENV)
ELSE IF T(X, DELETE) THEN ST := T1
ELSE (Q IS λY STREQUAL(X STRING, Y);
    IF Q "ATOM" THEN
       (AC := INC ACV; ST := CONS(M(PROGATOMDEN, INC ACV), STV))
```

```
ELSE IF Q "REF" OR Q "NCREF" THEN
   (MR := AUG(MRV, S1); ST := CONS(M(EXPREFDEN, INC MRV UL), T1))
ELSE IF (Q "SET" OR Q "NCSET") AND T(S2, REFDEN) THEN
   IF T(S2, EXPREFDEN) THEN
      (MR := REPLACE(S2 ADDRESS, S1, MRV); ST := CONS(S1, T2))
   ELSE (CN := CONC((M APPLY, M DELETE), CNV);
      ST := CONC((S1, S2 SETF, S1), T2))
ELSE IF Q "VAL" AND T(S1, REFDEN) THEN
   IF T(S1, EXPREFDEN) THEN ST := CONS(MRV S1 ADDRESS, T1)
   ELSE (CN := CONS(M APPLY, CNV); ST := CONC((M ERRORDEN, S1 VALF), T1))
ELSE IF Q "COERCE" THEN
   IF T(S1, REFDEN) THEN CN := CONC((M(BASICFUNCTINST, "VAL"),
      M(BASICFUNCTINST, "COERCE")), CNV)
   ELSE ()
ELSE IF Q "GOTO" AND T(S1, LABELDEN) THEN
   IF T(S1, PROGLABELDEN) THEN (CN := UNITSEQ M(EXEC, S1 BODY);
      ST := (); EN := S1 ENVR; DM := S1 DUMP)
   ELSE GOTO ERROR
ELSE ST := CONS(
   IF Q "ISINTEGER" THEN T(S1, INTCLASS), T1
   ELSE IF Q "ISBOOLEAN" THEN T(S1, BOOLCLASS), T1
   ELSE IF Q "ISCHAR" THEN T(S1, CHARCLASS), T1
   ELSE IF Q "ISATOM" THEN T(S1, ATOMDEN), T1
   ELSE IF Q "ISFUNCTION" THEN T(S1, FUNCTDEN), T1
   ELSE IF Q "ISREF" THEN T(S1, REFDEN), T1
   ELSE IF Q "ISLABEL" THEN T(S1, LABELDEN), T1
   ELSE IF Q "IMPREF" AND T(S2, FUNCTDEN) AND T(S1, FUNCTDEN) THEN
      M(IMPREFDEN, S2, S1), T2
   ELSE IF Q "EQUAL" OR Q "NCEQUAL" THEN
      T(S2, INTCLASS) AND T(S1, INTCLASS) AND S2 = S1
      OR T(S2, BOOLCLASS) AND T(S1, BOOLCLASS) AND S2 = S1
      OR T(S2, CHARCLASS) AND T(S1, CHARCLASS) AND S2 = S1
      OR T(S2, LLDEN) AND T(S1, LLDEN) OR T(S2, ULDEN) AND T(S1, ULDEN)
      OR T(S2, PROGATOMDEN) AND T(S1, PROGATOMDEN) AND S2 NAME = S1 NAME
      OR T(S2, EXPREFDEN) AND T(S1, EXPREFDEN) AND S2 ADDRESS = S1 ADDRESS, T2
   ELSE IF Q "GREATER" AND T(S2, INTCLASS) AND T(S1, INTCLASS) THEN
      GREATER(S2, S1), T2
   ELSE IF Q "CHARGREATER" AND T(S2, CHARCLASS) AND T(S1, CHARCLASS) THEN
      CHARGREATER(S2, S1), T2
   ELSE IF Q "INC" AND T(S1, INTCLASS) THEN INC S1, T1
   ELSE IF Q "DEC" AND T(S1, INTCLASS) THEN DEC S1, T1
   ELSE IF Q "READCHAR" THEN READCHAR(), STV
   ELSE IF Q "WRITECHAR" AND T(S1, CHARCLASS) THEN (WRITECHAR S1; S1, T1)
   ELSE GOTO ERROR));
M(STATE, VAL CN, VAL ST, VAL EN, VAL DM, VAL MR, VAL AC)));
```

To complete the definition of the interpreter, we must give the sequences of predefined identifier strings and value denotations which are used by GETVALPREDEF:

```
    PREDEFIDSTRS IS ("ISINTEGER", "ISBOOLEAN", "ISCHAR", "ISATOM", "ISFUNCTION",
    "ISREF", "ISLABEL", "ATOM", "NCREF", "REF", "IMPREF", "NCSET", "SET", "VAL",
    "COERCE", "GOTO", "NCEQUAL", "EQUAL", "GREATER", "CHARGREATER", "INC", "DEC",
    "READCHAR", "WRITECHAR", "TRUE", "FALSE", "QUOTECHAR", "LL", "UL", "ERROR");
```

Since the value of PREDEFVDENS is fairly complex, we give an expression for computing it rather than an explicit value:

```
     PREDEFVDENS IS (X IS CRIDENT(); C IS M(BASICFUNCTINST, "COERCE");
     ONE IS M(CONSTANT, 1); TWO IS M(CONSTANT, 2);
     P1 IS UNITSEQ M DELETE; P2 IS (); P3 IS UNITSEQ C;
     P4 IS (M(BIND, X), X, C, ONE, M APPLY, X, C, TWO, M APPLY);
     P5 IS (M(BIND, X), X, C, ONE, M APPLY, X, C, TWO, M APPLY, C);
     P6 IS (M(BIND, X), X, C, ONE, M APPLY, C, X, C, TWO, M APPLY, C);
     PLIST IS(P3, P3, P3, P3, P3, P2, P3, P1, P2, P3, P6, P4, P5, P2, P2, P3,
         P4, P6, P6, P6, P3, P3, P1, P3);
     D IS (TRUE, FALSE, QUOTECHAR, M LLDEN, M ULDEN, M ERRORDEN);
     CONC(VECTOR(1, 24, λI M(FUNCTDEN,
         AUG(PLIST I, M(BASICFUNCTINST, PREDEFIDSTRS I)), ())), D) );
```

The denotation of each basic function has an empty environment field and a control field consisting of the appropriate basic function instruction preceded by an instruction sequence called a <u>prelude</u>, whose purpose is to spread and coerce the function arguments. The six preludes shown above handle the following cases:

P1 - no arguments, P2 - one noncoerced argument, P3 - one coerced argument, P4 - two noncoerced arguments, P5 - two arguments, the second of which is coerced, P6 - two coerced arguments.

A Direct Interpreter

In the preceding definition of an interpreter for GEDANKEN, we have largely avoided the type of definitional circularity in which some language feature is defined by using the same feature in the interpreter itself. In the few cases where such circularity occurs, the feature involved either has a commonly accepted unambiguous meaning (e.g., integer or boolean arithmetic), or has a machine-dependent aspect whose precise definition we wish to avoid (e.g., character arithmetic or the input-output facilities provided by READCHAR and WRITECHAR).

In this section we present a second interpreter, called DINTERPRET, which has been designed to maximize, rather than minimize, definitional circularity. It must be emphasized that DINTERPRET is not a definition of GEDANKEN; indeed a surprising variety of changes in the semantics of the language leave the validity of DINTERPRET unchanged. Nevertheless, DINTERPRET can be regarded as an important (albeit unproved) theorem about GEDANKEN, and its brevity is a measure of the simplicity of the language.

The transition from INTERPRET to DINTERPRET is based on two changes:

(1) Instead of defining an abstract syntax for value denotations, we assume that the denotation of any member of the universal value set is an equivalent value of the same type. Thus, for example, a function or label value of the program being interpreted will be denoted by an equivalent function or label value of the interpreter itself. To effect this change, we must replace the class definition of constants by

```
    (CLASS, CONSTANT, (VALUE, UNIVERSAL))
```

and replace the list of value denotations for predefined identifiers by

```
    PREDEFVDENS IS (ISINTEGER, ISBOOLEAN, ISCHAR, ISATOM, ISFUNCTION,
    ISREF, ISLABEL, ATOM, NCREF, REF, IMPREF, NCSET, SET, VAL,
    COERCE, GOTO, NCEQUAL, EQUAL, GREATER, CHARGREATER, INC, DEC,
    READCHAR, WRITECHAR, TRUE, FALSE, QUOTECHAR, LL, UL, ERROR);
```

(2) Instead of treating an environment as a sequence of marks and identifier-value pairs, we will take the more direct approach of defining an environment to be a function which maps identifiers into the denotations of the values to which they are bound.

Then

```
DINTERPRET ISR λX EVAL(X, GETVALPREDEF);
```

where EVAL(X, E) computes the value of the expression X in the environment E:

```
EVAL ISR λ(X, E)
IF T(X, CONSTANT) THEN X VALUE
ELSE IF T(X, IDENT) THEN E X
ELSE IF T(X, FUNCTDES) THEN (EVAL(X FUNCTPART, E)) EVAL(X ARGPART, E)
ELSE IF T(X, LAMBDAEXP) THEN
    λA EVAL(X BODY, λK IF IDEQUAL(K, X PARAMPART) THEN A ELSE E K)
ELSE IF T(X, CONDEXP) THEN
    IF EVAL(X PREMISS, E) THEN EVAL(X CONCLUSION, E) ELSE EVAL(X ALTERNATIVE,
ELSE IF T(X, CASEEXP) THEN (I IS EVAL(X INDEX, E);
    IF ISATOM I THEN (X BODY) I ELSE EVAL((X BODY) I, E))
ELSE (R IS X RDECLPART; L IS X LDECLPART; S IS REF X BODY;
    NE ISR λK SEARCH(L UL, λI IDEQUAL(K, (L I) LEFT),
        λI (GOTO L2; L1: S := (L I) RIGHT; GOTO L3; L2: L1),
        λ() SEARCH(R UL, λI IDEQUAL(K, (R I) LEFT),
            λI λA  EVAL(((R I) RIGHT) BODY,
                λK IF IDEQUAL(K, ((R I) RIGHT) PARAMPART) THEN A ELSE NE K),
            λ() E K));
    L3: EVALSEQ(VAL S, NE));
EVALSEQ ISR λ(X, E)
    IF X UL = 1 THEN EVAL(X 1, E) ELSE (EVAL(X 1, E); EVALSEQ(TAIL X, E));
```

IV. POSSIBLE EXTENSIONS AND MODIFICATIONS

Hopefully, the basic principles underlying GEDANKEN will eventually be applicable to the design of an efficient and practically useful programming language. We conclude by discussing some of the research problems that must be solved to reach this goal.

Type Declarations

To achieve efficient data representation, the programmer must be able to define sets of values, and specify that the range of various identifiers, function results, and references are to be limited to such sets. Such information also allows a variety of programming errors to be detected during compilation.

Probably the most natural approach is an extension of Hoare's concept of record classes.(6) The programmer would be able to declare an arbitrary number of disjoint function, reference, and label classes. He would then be required to specify the range of each identifier, function result, or reference to be some union of such classes (and/or predefined classes of primitive data). Presumably, all functions in the same class would have the same argument and result ranges, and all references in the same class would have the same value range.

Class declarations would be permitted in the head of any block. If these declarations are assumed to define a distinct class each time the block is executed, then the language can be arranged so that all values in a given class must become inaccessible when the block activation in which the class was defined becomes inaccessible. Hopefully, this situation can be exploited to increase the efficiency of storage allocation.

The functional approach to data structures used in GEDANKEN places special requirements on a type declaration facility. In particular, if an inhomogeneous data structure such as a record is to be treated as a function, then in declaring the range of such a function it must be possible to specify a dependency on the function argument. Thus, for example, the set of lists of integers would be the union of the set {NIL} with a class of functions with domain {1, 2} which mapped 1 into an integer and 2 into a list of integers.

A final problem is the need for a more flexible set specification than unions of classes. For example, if "matrix" is a function class and ADDMAT(X, Y) is a function which adds matrices, then it should be possible to specify, not only that X, Y and the result of ADDMAT are matrices, but also that these matrices must have the same row and column dimensions.

Open Functions

An efficient compilation of GEDANKEN programs which manipulate complex data structures will require that certain function designators should be replaced by modified copies of the corresponding function body, and that these copies should then be simplified to take advantage of constant arguments. Typical examples are the record-manipulation functions T and M defined in the preceding chapter.

The ability to produce open code of this sort could be provided by adding a macro-definitional facility to the language. A second approach, more in keeping with the spirit of the language, would be to permit certain lambda expressions to be given an OPEN attribute.

This raises the question of whether a compiler could determine automatically when a designator of a lambda-defined function should be replaced by a copy of the function body. Until recently, the author believed that such an expansion could be performed for any function which was defined by a nonrecursive declaration. Unfortunately, this conjecture is disproved by the existence of the following nonrecursive _fixed-point function_:

```
Y IS λG (U IS λV G(λX (V V) X); U U);
```

which can be used to convert any simply recursive function (i.e., a function which calls itself directly but not indirectly via other functions) into an equivalent nonrecursive function.(15)

Thus suppose a recursive function F is defined by F ISR b, where F is the only identifier which occurs free in b. Let F1 be the nonrecursive function defined by F1 IS λF (b). Then the function (Y F1) can be shown to be equivalent to F, with the same domain of termination. Moreover, the expansion of a function designator such as (Y F1) X by repeated substitution of the definitions of Y and F1 will never terminate.

Label Value Difficulties

The properties of label values in GEDANKEN have certain potentially unfortunate consequences.

The use of label-valued references can frequently cause the preservation of data which will no longer be accessed by a computation. If L is a label-valued reference, then GOTO L will cause execution to proceed from the control status denoted by L. But the unchanged control status (i.e., environment and dump) must also be saved in case GOTO L is executed again before the value of L is changed. If, in fact, such a repeated jump cannot occur, then information will be saved unnecessarily unless the programmer goes to the trouble of resetting L immediately after the original jump. (As an example, the program given in Chapter II for linking the coroutines COMPILE and ASSEMBLE will preserve the control status of these routines unnecessarily.)

Presumably, it would be better to force the programmer to extra trouble in order to preserve, rather than discard, a reactivated control status. This might be accomplished by adapting the concept of "process" used in simulation languages, and providing a basic function for copying processes. However, it is not clear how to combine the process concept with an ALGOL-like use of label values in a clean manner which does not violate the principle of completeness.

A second difficulty is the inability of a label value to preserve the values of references as part of its control status. In the nondeterministic parser described in Chapter II, the restriction on the use of references in the function PARSE arises from this problem.

Finally, the use of label values introduces serious problems if a strict order of evaluation is not imposed on a GEDANKEN program. To permit code optimization, it is desirable to allow the independent subexpressions of a compound expression (such as a sequence expression) to be evaluated in any order, or even to have the steps of their evaluation intermixed. This can be done in the applicative subset of GEDANKEN without affecting the results of any program. The introduction of references makes the results indeterminate if expressions with interfering side effects are executed in parallel.

But when label values are introduced, the number of times various expressions are evaluated also becomes indeterminate. Even the simple program

```
        (X IS REF 0; (X := INC X, GOTO L); L: VAL X)
```

could produce either zero or one. The use of label-valued references leads to more paradoxical programs, such as

```
        (X IS REF 0; L IS REF 0; M IS REF 0; L := L1;
             (X := INC X, (M := M1; M1: GOTO L));
             L1: L := L2; GOTO M; L2: VAL X)
```

which might produce zero, one, or two.

APPENDIX: BASIC FUNCTIONS IN GEDANKEN

Predefined Identifier	Defined in Interpreter	Number of Arguments	Argument Coercion	Types of Arguments	Type of Result
ISINTEGER	yes	1	yes	any	Boolean
ISBOOLEAN	yes	1	yes	any	Boolean
ISCHAR	yes	1	yes	any	Boolean
ISATOM	yes	1	yes	any	Boolean
ISFUNCTION	yes	1	yes	any	Boolean
ISREF	yes	1	no	any	Boolean
ISLABEL	yes	1	yes	any	Boolean
ATOM	yes	0	--	--	atom
UNITSEQ	no	1	no	any	function
VECTOR	no	3	yes	integer, integer, function	function
NCREF	yes	1	no	any	reference
REF	yes	1	yes	any	reference
IMPREF	yes	2	yes	function, function	reference
NCSET	yes	2	no	reference, any	any
SET	yes	2	2nd arg	reference, any	any
VAL	yes	1	no	reference	any
COERCE	yes	1	--	any	any except reference
GOTO	yes	1	yes	label value	--
NCEQUAL	yes	2	no	any, any	Boolean
EQUAL	yes	2	yes	any, any	Boolean
GREATER	yes	2	yes	integer, integer	Boolean
CHARGREATER	yes	2	yes	character, character	Boolean
INC	yes	1	yes	integer	integer
DEC	yes	1	yes	integer	integer
NEG	no	1	yes	integer	integer
ADD	no	2	yes	integer, integer	integer
SUBTRACT	no	2	yes	integer, integer	integer
MULTIPLY	no	2	yes	integer, integer	integer
DIVIDE	no	2	yes	integer, integer $\neq 0$	integer
REMAINDER	no	2	yes	integer, integer $\neq 0$	integer
NOT	no	1	yes	Boolean	Boolean
INTTODIGIT	no	1	yes	$0 \leq$ integer ≤ 9	character
DIGITTOINT	no	1	yes	character	integer
READCHAR	yes	0	--	--	character
WRITECHAR	yes	1	yes	character	character

ACKNOWLEDGEMENTS

The author wishes to thank Dr. M. D. MacLaren of Argonne National Laboratory and Professor Arthur Evans, Jr., of MIT for their stimulating discussions and helpful suggestions.

REFERENCES

1. van Wijngaarden, A. (Ed.), Mailloux, B.J., Peck, J.E.L., and Koster, C.H.A. Draft report on the algorithmic language ALGOL 68. MR 93, Mathematisch Centrum, Amsterdam, January, 1968.
2. Naur, P. (Ed.) Revised report on the algorithmic language ALGOL 60. Comm. ACM, 6 (January, 1963), 1-17.
3. McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, Part I. Comm. ACM, 3 (April, 1960), 184-195.

   McCarthy, J. et al. LISP 1.5 programmers manual. MIT Press, Cambridge, Mass., 1962.
4. Wirth, N., and Weber, H. EULER - a generalization of ALGOL, and its formal definition: Part I, Part II. Comm. ACM, 9 (January, February, 1966), 13-25, 89-99.
5. Evans, A. PAL - a language designed for teaching programming linguistics. Proc. ACM 23rd Natl. Conf. 1968, pp. 395-403.
6. Wirth, N., and Hoare, C.A.R. A contribution to the development of ALGOL. Comm. ACM, 9 (June, 1966), 413-432.
7. Dahl, O.J., and Nygaard K. SIMULA - An ALGOL-based simulation language. Comm. ACM, 9 (September, 1966), 671-678.

   Dahl, O.J., Myhrhaug, B., and Nygaard, K. SIMULA 67 common base language. Publ. No. S-2, Norwegian Computing Center, Oslo, May, 1968.
8. Floyd, R.W. Nondeterministic algorithms. J. ACM, 14 (October, 1967), 636-644.
9. Reynolds, J.C. An introduction to the COGENT programming system. Proc. ACM 20th Natl. Conf., 1965, pp. 422-436.

   Reynolds, J.C. COGENT programming manual. ANL-7022, Argonne National Laboratory, Argonne, Illinois, March, 1965.
10. Earley, J.C. An efficient context-free parsing algorithm. Carnegie-Mellon University, Pittsburgh, Pa., August, 1968.

11. Lucas, P., Lauer, P., and Stigleitner, H. Method and notation for the formal definition of programming languages. TR 25.087, IBM Laboratory Vienna, June, 1968.

12. Landin, P. J. A correspondence between ALGOL 60 and Church's lambda-notation, Part I, Part II. Comm. ACM, 8 (February, March, 1965), 89-101, 158-165.

13. McCarthy, J. Towards a mathematical science of computation. Information Processing 62 (IFIP Congress), Popplewell, C. M. (Ed.), North-Holland Publishing Co., Amsterdam, 1963, pp. 21-28.

14. Irons, E. T. A syntax directed compiler for ALGOL 60. Comm ACM, 4, (January, 1961), 51-55.

15. Evans, A. Private communication.

Morris, J. H. Lambda-calculus models of programming languages. MAC-TR-57, Project MAC, MIT, Cambridge, Mass., December, 1968.

ARGONNE NATIONAL LAB WEST
3 4444 00008272 7